Dot Net Technology

BCA
Fifth
SEMESTER

Mechi Multiple Campus

Bhadrapur, Jhapa

Notes Prepared By
Raju Poudel
Lecturer, MMC

Unit -1 Introduction to C# and .NET Framework [7 Hrs]

Object Orientation; Type Safety; Memory Management; Platform Support; C# and CLR; CLR and .NET Framework; Framework Overview; .NET Standard 2.0; Applied Technologies

Introduction to C# Language

C# is a general-purpose, object-oriented programming language. **C# was developed by Anders Hejlsberg and his team during the development of .Net Framework.** It can be used to create desktop, web and mobile applications.

C# is a hybrid of C and C++; it is a Microsoft programming language developed to compete with Sun's Java language. C# is an object-oriented programming language used with XML-based Web services on the .NET platform and designed for improving productivity in the development of Web applications.

C# is an elegant and type-safe object-oriented language that enables developers to build a variety of secure and robust applications that run on the .NET Framework. You can use C# to create Windows client applications, XML Web services, distributed components, client-server applications, database applications, and much, much more. Visual C# provides an advanced code editor, convenient user interface designers, integrated debugger, and many other tools to make it easier to develop applications based on the C# language and the .NET Framework.

As an object-oriented language, C# supports the concepts of encapsulation, inheritance, and polymorphism. All variables and methods, including the Main method, the application's entry point, are encapsulated within class definitions.

The following reasons make C# a widely used professional language (Features of C#):

1. It is a modern, general-purpose programming language
2. It is object oriented.
3. It is component oriented.
4. It is easy to learn.
5. It is a structured language.
6. It produces efficient programs.
7. It can be compiled on a variety of computer platforms.
8. It is a part of .Net Framework.

Object Orientation

C# is a rich implementation of the object-orientation paradigm, which includes encapsulation, abstraction, inheritance, and polymorphism. Encapsulation means creating a boundary around an object, to separate its external (public) behaviour from its internal (private) implementation details.

The distinctive features of C# from an object-oriented perspective are:

Unified type system

The fundamental building block in C# is an encapsulated unit of data and functions called a type. C# has a unified type system, where all types ultimately share a common base type. This means that all types, whether they represent business objects or are primitive types such as numbers, share the same basic functionality. For example, an instance of any type can be converted to a string by calling its ToString method.

Classes and interfaces

In a traditional object-oriented paradigm, the only kind of type is a class. In C#, there are several other kinds of types, one of which is an interface. An interface is like a class, except that it only describes members. The implementation for those members comes from types that implement the interface. Interfaces are particularly useful in scenarios where multiple inheritance is required (unlike languages such as C++ and Eiffel, C# does not support multiple inheritance of classes).

Properties, methods, and events

In the pure object-oriented paradigm, all functions are methods. In C#, methods are only one kind of function member, which also includes properties and events. Properties are function members that encapsulate a piece of an object's state, such as a button's colour or a label's text. Events are function members that simplify acting on object state changes.

While C# is primarily an object-oriented language, it also borrows from the functional programming paradigm. Specifically:

Functions can be treated as values

Through the use of delegates, C# allows functions to be passed as values to and from other functions.

C# supports patterns for purity

Core to functional programming is avoiding the use of variables whose values change, in favour of declarative patterns. C# has key features to help with those patterns, including the ability to write unnamed functions on the fly that "capture" variables (lambda expressions), and the ability to perform list or reactive programming via query expressions. C# also makes it easy to define read-only fields and properties for writing immutable (read-only) types.

Type Safety

C# is primarily a type-safe language, meaning that instances of types can interact only through protocols they define, thereby ensuring each type's internal consistency. For instance, C# prevents you from interacting with a string type as though it were an integer type.

More specifically, C# supports static typing, meaning that the language enforces type safety at compile time. This is in addition to type safety being enforced at run- time.

Static typing eliminates a large class of errors before a program is even run. It shifts the burden away from runtime unit tests onto the compiler to verify that all the types in a program fit together correctly. This makes large programs much easier to manage, more predictable, and more robust. Furthermore, static typing allows tools such as IntelliSense in Visual Studio to help you write a program, since it knows for a given variable what type it is, and hence what methods you can call on that variable.

C# is also called a strongly typed language because its type rules (whether enforced statically or at runtime) are very strict. For instance, you cannot call a function that's

designed to accept an integer with a floating-point number, unless you first explicitly convert the floating-point number to an integer. This helps prevent mistakes.
Strong typing also plays a role in enabling C# code to run in a sandbox—an environment where every aspect of security is controlled by the host. In a sandbox, it is important that you cannot arbitrarily corrupt the state of an object by bypassing its type rules.

Memory Management

C# relies on the runtime to perform automatic memory management. The Com- mon Language Runtime has a garbage collector that executes as part of your program, reclaiming memory for objects that are no longer referenced. This frees programmers from explicitly deallocating the memory for an object, eliminating the problem of incorrect pointers encountered in languages such as C++.
C# does not eliminate pointers: it merely makes them unnecessary for most programming tasks.

Platform Support

Historically, C# was used almost entirely for writing code to run on Windows platforms. Recently, however, Microsoft and other companies have invested in other platforms, including Linux, macOS, iOS, and Android.
Xamarin™ allows cross platform C# development for mobile applications, and Portable Class Libraries are becoming increasingly widespread.
Microsoft's ASP.NET Core is a cross-platform lightweight web hosting framework that can run either on the .NET Framework or on .NET Core, an open source cross-platform runtime.

C# and CLR

C# depends on a runtime equipped with a host of features such as automatic memory management and exception handling. **At the core of the Microsoft .NET Framework is the Common Language Runtime (CLR), which provides these runtime features**. (The .NET Core and Xamarin frameworks provide similar runtimes.)

The CLR is language-neutral, allowing developers to build applications in multiple languages (e.g., C#, F#, Visual Basic .NET, and Managed C++).

C# is one of several managed languages that get compiled into managed code. Man- aged code is represented in Intermediate Language or IL. The CLR converts the IL into the native code of the machine, such as X86 or X64, usually just prior to execution. This is referred to as Just-In-Time (JIT) compilation. Ahead-of-time compilation is also available to improve start up time with large assemblies or resource constrained devices (and to satisfy iOS app store rules when developing with Xamarin).

The container for managed code is called an assembly or portable executable. An assembly can be an executable file (.exe) or a library (.dll), and contains not only IL, but type information (metadata). The presence of metadata allows assemblies to reference types in other assemblies without needing additional files.
A program can query its own metadata (reflection), and even generate new IL at runtime (reflection.emit)

Introduction and Overview of .Net Framework

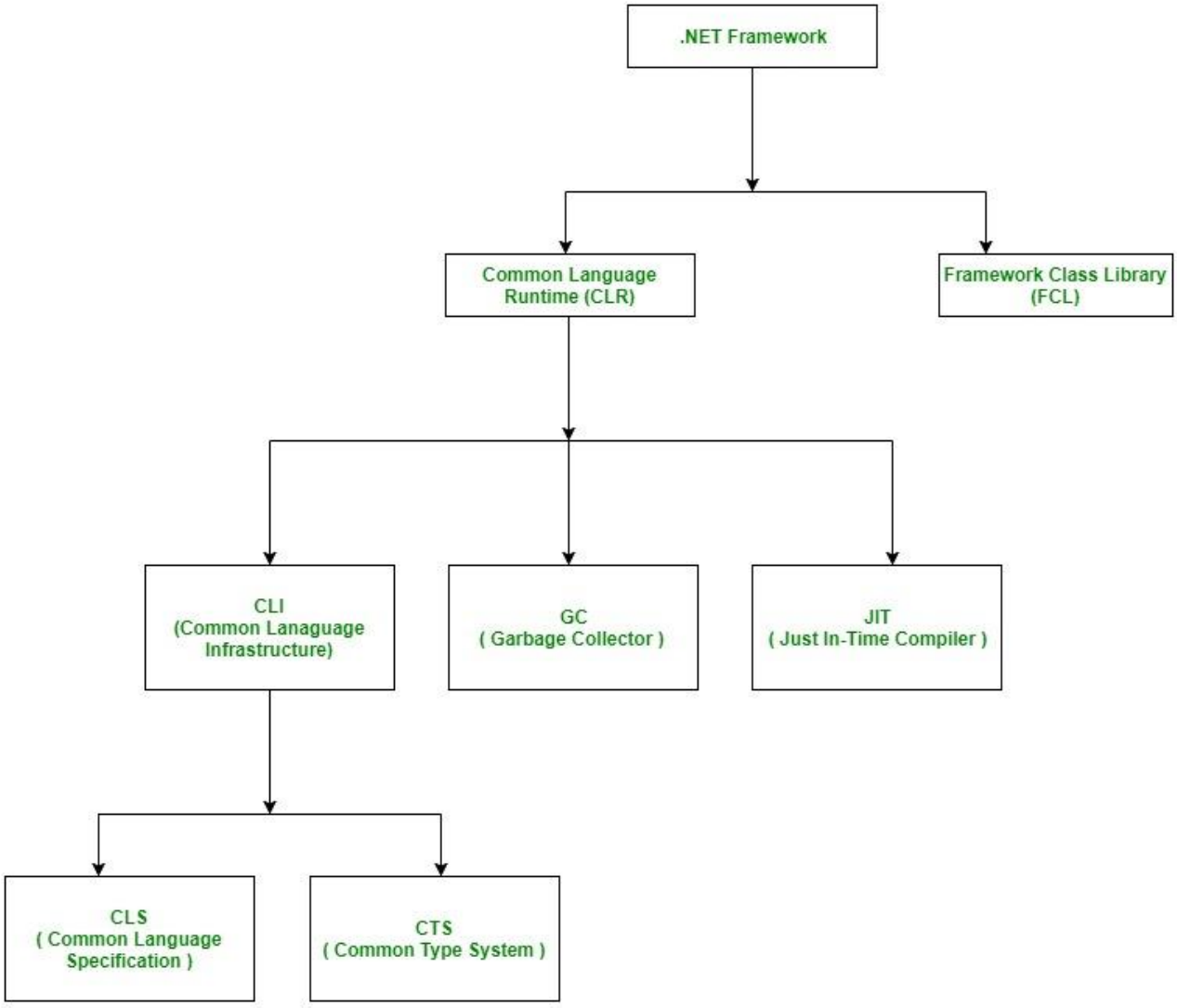
.NET is the framework for which we develop applications. It sits in between our application programs and operating system.

.NET provides an object oriented environment. It ensures safe execution of the code by performing required runtime validations. For example, it is never possible to access an element of an array outside the boundary. Similarly, it is not possible to a program to write into another programs area, etc. The runtime validations performed by .NET makes the entire environment robust.

It is a programming infrastructure created by Microsoft for building, deploying, and running applications and services that use .NET technologies, such as desktop applications and Web services.

The .NET Framework consists of:

- the Common Language Runtime
- the Framework Class Library



Common Language Runtime(CLR):

CLR is the basic and Virtual Machine component of the .NET Framework. It is the run-time environment in the .NET Framework that runs the codes and helps in making the development process easier by providing the various services such as remoting, thread management, type-safety, memory management, robustness etc. Basically, it is responsible for managing the execution of .NET programs regardless of any .NET programming language. It also helps in the management of code, as code that targets the runtime is known as the Managed Code and code doesn't target to runtime is known as Unmanaged code.

Features of Common Language Runtime:

1. The runtime enforces code access security. For example, users can trust that an executable embedded in a Web page can play an animation on screen or sing a song, but cannot access their personal data, file system, or network.
2. The runtime also enforces code robustness by implementing a strict type-and-code-verification infrastructure called the common type system (CTS). The CTS ensures that all managed code is self-describing.
3. The runtime also accelerates developer productivity. For example, programmers can write applications in their development language of choice, yet take full advantage of the runtime, the class library, and components written in other languages by other developers. Any compiler vendor who chooses to target the runtime can do so. Language compilers that target the .NET Framework make the features of the .NET Framework available to existing code written in that language, greatly easing the migration process for existing applications.
4. While the runtime is designed for the software of the future, it also supports software of today and yesterday.
5. The runtime is designed to enhance performance.
6. The runtime can be hosted by high-performance, server-side applications, such as Microsoft SQL Server and Internet Information Services (IIS).

Framework Class Library (FCL):

The class library is a comprehensive, object-oriented collection of reusable types that you can use to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services.
It is the collection of reusable, object-oriented class libraries and methods etc. that can be integrated with CLR. Also called the Assemblies. It is just like the header files in C/C++ and packages in the java. Installing .NET framework basically is the installation of CLR and FCL into the system.

The core libraries are sometimes collectively called the Base Class Library (BCL). The entire framework is called the Framework Class Library (FCL).

Features of Framework Class Library:

An object-oriented class library, the .NET Framework types enable you to accomplish a range of common programming tasks, including tasks such as string management, data collection, database connectivity, and file access. In addition to these common tasks, the

class library includes types that support a variety of specialized development scenarios. For example, you can use the .NET Framework to develop the following types of applications and services:

1. Console applications
2. Windows GUI applications (Windows Forms).
3. Windows Presentation Foundation (WPF) applications.
4. ASP.NET applications.
5. Windows services.
6. Service-oriented applications using Windows Communication Foundation (WCF).
7. Workflow-enabled applications using Windows Workflow Foundation (WF).

What's New in .NET Framework 4.6

- The Garbage Collector (GC) offers more control over when (not) to collect via new methods on the GC class. There are also more fine-tuning options when calling `GC.Collect`.
- There's a brand-new faster 64-bit JIT compiler.
- The `System.Numerics` namespace now includes hardware-accelerated matrix, vector types, `BigInteger` and `Complex`.
- There's a new `System.AppContext` class, designed to give library authors a consistent mechanism for letting consumers switch new API features in or out.
- `Tasks` now pick up the current thread's culture and UI culture when created.
- More collection types now implement `IReadOnlyCollection<T>`.
- WPF has further improvements, including better touch and high-DPI handling.
- ASP.NET now supports HTTP/2 and the Token Binding Protocol in Windows 10.

What's New in .NET Framework 4.7

Framework 4.7 is more of a maintenance release than a new-feature release, with numerous bug fixes and minor improvements. Additionally:

- The `System.ValueTuple` struct is part of Framework 4.7, so you can use tuples in C# 7 without referencing the *System.ValueTuple.dll* assembly.
- WPF has better touch support.
- Windows Forms has better support for high-DPI monitors.

Table 5-1 shows the history of compatibility between each version of C#, the CLR, and the .NET Framework.

Table 5-1. C#, CLR, and .NET Framework versions

C# version	CLR version	.NET Framework versions
1.0	1.0	1.0
1.2	1.1	1.1
2.0	2.0	2.0, 3.0
3.0	2.0 (SP2)	3.5
4.0	4.0	4.0
5.0	4.5 (Patched CLR 4.0)	4.5
6.0	4.6 (Patched CLR 4.0)	4.6
7.0	4.6/4.7 (Patched CLR 4.0)	4.6/4.7

Other Frameworks

The Microsoft .NET Framework is the most expansive and mature framework, but runs only on Microsoft Windows (desktop/server). Over the years, other frame- works have emerged to support other platforms. There are currently three major players besides the .NET Framework, all of which are currently owned by Microsoft:

Universal Windows Platform (UWP)

For writing Windows 10 Store Apps and for targeting Windows 10 devices (mobile, XBox, Surface Hub, Hololens). Your app runs in a sandbox to lessen the threat of malware, prohibiting operations such as reading or writing arbitrary files.

.NET Core with ASP.NET Core

An open source framework (originally based on a cut-down version of the .NET Framework) for writing easily deployable Internet apps and micro services that run on Windows, macOS, and Linux. Unlike the .NET Frame- work, .NET Core can be packaged with the web application and xcopy deployed (self-contained deployment).

Xamarin

For writing mobile apps that target iOS, Android, and Windows Mobile. The Xamarin company was purchased by Microsoft in 2016.

Table 1-1 compares the current platform support for each of the major frameworks

Table 1-1. Platform support for the popular frameworks

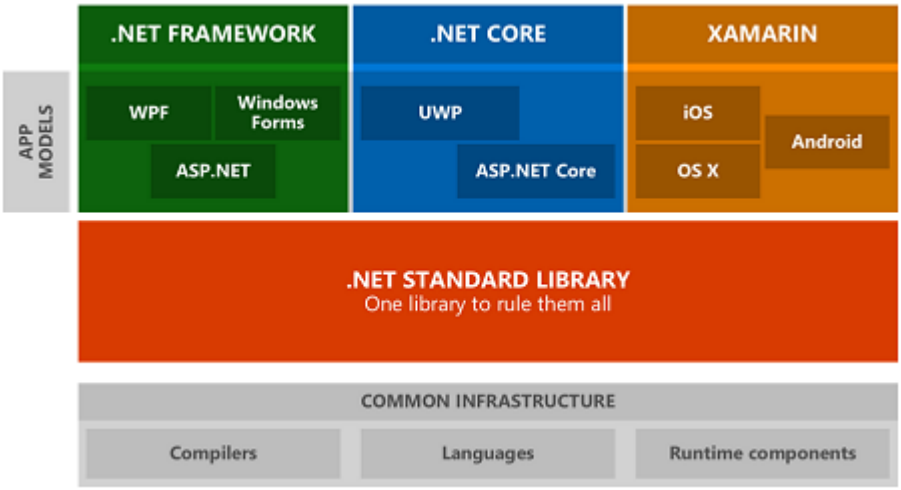
Target operating system	.NET Framework	UWP	.NET Core	Xamarin
Windows 7/8	Yes		Yes	
Windows 10 desktop/server	Yes	Yes	Yes	
Windows 10 devices		Yes		Yes
Linux			Yes	
macOS			Yes	
iOS (iPhone)				Yes
Android				Yes

.Net Standard 2.0

.Net Standard is a specification which dictates what the Base Class Libraries of different .Net platforms should implement to unify the Base Class Libraries of different .Net Platforms. Here, Platform means full .Net Framework, .Net Core, Xamarin, Silverlight, XBox etc. This also enables code sharing between applications that runs on these different platforms. For example, a library or a component that is developed on top of a platform that implements specific .Net Standard version can be shared by all the applications that runs on any of the platforms that implements same .Net Standard version.

.NET Standard is not a Framework; it's merely a specification describing a minimum baseline of functionality (types and members), which guarantees compatibility with a certain set of frameworks. The concept is similar to C# interfaces: .NET Standard is like an interface that concrete types (frameworks) can implement

Net Standard has solved all this in a different way, it provided an API specification which all platforms should implement to remain .Net Standard complaint. This has unified the base class libraries of different .Net platforms and has paved way to share libraries and also brought the BCL evolution centralized as seen below.



.NET Standard 2.0 is a new version that significantly increases the number of APIs compared to the previous version (1.6.1). In fact, the API surface has more than doubled with .NET Standard 2.0.

The .NET Standard 2.0 is supported by the following .NET implementations:

- .NET Core 2.0 or later
- .NET Framework 4.6.1 or later
- Mono 5.4 or later
- Xamarin.iOS 10.14 or later
- Xamarin.Mac 3.8 or later
- Xamarin.Android 8.0 or later
- Universal Windows Platform 10.0.16299 or later

What's new in the .NET Standard 2.0?

The .NET Standard 2.0 includes the following new features:

1) A vastly expanded set of APIs
2) Support for .NET Framework libraries
3) Support for Visual Basic
4) Tooling support for .NET Standard libraries

Older .NET Standards

There are also older .NET Standards in use, most notably 1.1, 1.2, 1.3, and 1.6. A higher-numbered standard is always a strict superset of a lower-numbered standard. For instance, if you write a library that targets .NET Standard 1.6, you will support not only recent versions of the four major frameworks, but also .NET Core 1.0. And if you target .NET Standard 1.3, you support everything we've already mentioned plus .NET Framework 4.6.0 (see Table 5-2).

Table 5-2. Older .NET Standards

If you target...	You also support...
Standard 1.6	.NET Core 1.0
Standard 1.3	Above plus .NET 4.6.0
Standard 1.2	Above plus .NET 4.5.1, Windows Phone 8.1, WinRT for Windows 8.1
Standard 1.1	Above plus .NET 4.5.0, Windows Phone 8.0, WinRT for Windows 8.0

The 1.x standards lack thousands of APIs that are present in 2.0, including much of what we describe in this book. This can make targeting a 1.x standard significantly more challenging, especially if you need to integrate existing code or libraries.

If you need to support older frameworks but don't need cross platform compatibility, a better option is to target an older version of a specific framework. In the case of Windows, a good choice is .NET Framework 4.5 because it's widely deployed (pre-installed on all machines running Windows 8 and later), and it contains most of what's in .NET Framework 4.7.

You can also think of .NET Standard as a lowest common denominator. In the case of .NET Standard 2.0, the four frameworks that implement it have a similar Base Class Library, so the lowest common denominator is big and useful. However, if you also want compatibility with .NET Core 1.0 (with its significantly cut-down BCL), the lowest common denominator—.NET Standard 1.x—becomes much smaller and less useful.

The CLR and Core Framework

System Types

The most fundamental types live directly in the System namespace. These include C#'s built-in types, the Exception base class, the Enum, Array, and Delegate base classes, and Nullable, Type, DateTime, TimeSpan, and Guid. The System namespace also includes types for performing mathematical functions (Math), generating random numbers (Random), and converting between various types (Convert and Bit Converter).

Text Processing

The System.Text namespace contains the StringBuilder class (the editable or mutable cousin of string), and the types for working with text encodings, such as UTF-8 (Encoding and its subtypes).
The System.Text.RegularExpressions namespace contains types that perform advanced pattern-based search-and-replace operations.

Collections

The .NET Framework offers a variety of classes for managing collections of items. These include both list- and dictionary-based structures, and work in conjunction with a set of standard interfaces that unify their common characteristics. All collection types are defined in the following namespaces:

```
System.Collections                // Nongeneric collections
System.Collections.Generic        // Generic collections
System.Collections.Specialized    // Strongly typed collections
System.Collections.ObjectModel    // Bases for your own collections
System.Collections.Concurrent     // Thread-safe collection (Chapter 23)
```

Queries

Language Integrated Query (LINQ) was added in Framework 3.5. LINQ allows you to perform type-safe queries over local and remote collections (e.g., SQL Server tables). A big advantage of LINQ is that it presents a consistent querying API across a variety of domains. The essential types reside in the following namespaces, and are part of .NET Standard 2.0:

```
System.Linq                 // LINQ to Objects and PLINQ
System.Linq.Expressions     // For building expressions manually
System.Xml.Linq             // LINQ to XML
```

The full .NET Framework also includes the following,

```
System.Data.Linq            // LINQ to SQL
System.Data.Entity          // LINQ to Entities (Entity Framework)
```

XML

XML is used widely within the .NET Framework, and so is supported extensively. The XML name- spaces are:

```
System.Xml                // XmlReader, XmlWriter + the old W3C DOM
System.Xml.Linq           // The LINQ to XML DOM
System.Xml.Schema         // Support for XSD
System.Xml.Serialization  // Declarative XML serialization for .NET types
System.Xml.XPath          // XPath query language
System.Xml.Xsl            // Stylesheet support
```

Diagnostics

Diagnostics refers to .NET's logging and assertion facilities and describe how to interact with other processes, write to the Windows event log, and use performance counters for monitoring. The types for this are defined in and under System.Diagnostics. Windows-specific features are not part of .NET Standard, and are avail- able only in the .NET Framework.

Concurrency and Asynchrony

Many modern applications need to deal with more than one thing happening at a time. Since C# 5.0, this has become easier through asynchronous functions and high-level constructs such as tasks and task combinators. Types for working with threads and asynchronous operations are in the System.Threading and System.Threading.Tasks namespaces.

Streams and I/O

The Framework provides a stream-based model for low-level input/output. Streams are typically used to read and write directly to files and network connections, and can be chained or wrapped in decorator streams to add compression or encryption functionality. The .NET Stream and I/O types are defined in and under the System.IO namespace, and the WinRT types for file I/O are in and under Windows.Storage.

Networking

You can directly access standard network protocols such as HTTP, FTP, TCP/IP, and SMTP via the types in System.Net.

```
System.Net
System.Net.Http          // HttpClient
System.Net.Mail          // For sending mail via SMTP
System.Net.Sockets       // TCP, UDP, and IP
```

The latter two namespaces are unavailable to Windows Store applications if you're targeting Windows 8/8.1 (WinRT), but are available to Windows 10 Store apps (UWP) as part of the .NET Standard 2.0 contract. For WinRT apps, use third-party libraries for sending mail, and the WinRT types in Windows.Networking.Sockets for working with sockets.

Serialization

The Framework provides several systems for saving and restoring objects to a binary or text representation. Such systems are required for distributed application technologies,

such as WCF, Web Services, and Remoting, and also to save and restore objects to a file. The types for serialization reside in the following namespaces:

```
System.Runtime.Serialization
System.Xml.Serialization
```

Assemblies, Reflection, and Attributes

The assemblies into which C# programs compile comprise executable instructions (stored as intermediate language or IL) and metadata, which describes the pro- gram's types, members, and attributes. Through reflection, you can inspect this metadata at runtime, and do such things as dynamically invoke methods. With Reflection.Emit, you can construct new code on the fly.

```
System
System.Reflection
System.Reflection.Emit  // .NET Framework only
```

Dynamic Programming

Dynamic Language Runtime, which has been a part of the CLR since Framework 4.0. The types for dynamic programming are in System.Dynamic.

Security

Code access, role, and identity security, and the transparency model introduced in CLR 4.0. Cryptography can be done in the Framework, covering encryption, hashing, and data protection. The types for this are defined in:

System.Security
System.Security.Permissions
System.Security.Policy
System.Security.Cryptography

Advanced Threading

C#'s asynchronous functions make concurrent programming significantly easier because they lessen the need for lower-level techniques. However, there are still times when you need signaling constructs, thread-local storage, reader/writer locks, and so on. Threading types are in the System.Threading namespace.

Applied Technologies

Descriptions of .NET Implementations

	OS	Open Source	Purpose
.NET Framework	Windows	No	Used for building Windows desktop applications and ASP.NET Web apps running on IIS.
.NET Core	Windows, Linux, macOS	Yes	Used for building cross-platform console apps and ASP.NET Core Web apps and cloud services.
Xamarin	iOS, Android, macOS	Yes	Used for building mobile applications for iOS and Android, as well as desktop apps for macOS.
.NET Standard	N/A	Yes	Used for building libraries that can be referenced from all .NET implementations, such as .NET Framework, .NET Core and Xamarin.

Following are the different applied technologies:

User-Interface APIs

User-interface–based applications can be divided into two categories: thin client, which amounts to a website, and rich client, which is a program the end user must download and install on a computer or mobile device.

For thin client applications, .NET provides ASP.NET and ASP.NET Core.For rich-client applications that target Windows 7/8/10 desktop, .NET provides the WPF and Windows Forms APIs. For rich-client apps that target iOS, Android, and Windows Phone, there's Xamarin, and for writing rich-client store apps for Win- dows 10 desktop and devices.

Finally, there's a hybrid technology called Silverlight, which has been largely abandoned since the rise of HTML5.

ASP.NET

Applications written using ASP.NET host under Windows IIS and can be accessed from any web browser. Here are the advantages of ASP.NET over rich-client technologies:

- There is zero deployment at the client end.
- Clients can run a non-Windows platform.
- Updates are easily deployed.

In writing your web pages, you can choose between the traditional Web Forms and the newer MVC (Model-View-Controller) API. Both build on the ASP.NET infra- structure. Web Forms has been part of the Framework since its inception; MVC was written much later in response to the success of Ruby on Rails and MonoRail. It provides, in general, a better programming abstraction than Web Forms; it also allows more control over the generated HTML.

ASP.NET Core

A relatively recent addition, ASP.NET Core is similar to ASP.NET, but runs on both .NET Framework and .NET Core (allowing for cross-platform deployment). ASP.NET Core features a lighter-weight modular architecture, with the ability to self-host in a custom process, and an open source license. Unlike its predecessors, ASP.NET Core is not dependent on System.Web and the historical baggage of Web Forms. It's particularly suitable for micro-services and deployment inside containers.

Windows Presentation Foundation (WPF)

WPF was introduced in Framework 3.0 for writing rich-client applications. The benefits of WPF over its predecessor, Windows Forms, are as follows:

- It supports sophisticated graphics, such as arbitrary transformations, 3D rendering, multimedia, and true transparency. Skinning is supported through styles and templates.
- Its primary measurement unit is not pixel-based, so applications display correctly at any DPI (dots per inch) setting.
- It has extensive and flexible layout support, which means you can localize an application without danger of elements overlapping.
- Rendering uses DirectX and is fast, taking good advantage of graphics hard- ware acceleration.
- It offers reliable data binding.
- User interfaces can be described declaratively in XAML files that can be maintained independently of the "code-behind" files—this helps to separate appearance from functionality.

Windows Forms

Windows Forms is a rich-client API that's as old as the .NET Framework. Com- pared to WPF, Windows Forms is a relatively simple technology that provides most of the features you need in writing a typical Windows application. It also has significant relevancy in maintaining legacy applications. It has a number of drawbacks, though, compared to WPF:

- Controls are positioned and sized in pixels, making it easy to write applications that break on clients whose DPI settings differ from the developer's (although this has improved somewhat in Framework 4.7).
- The API for drawing nonstandard controls is GDI+, which, although reasonably flexible, is slow in rendering large areas (and without double buffering, may flicker).
- Controls lack true transparency.
- Most controls are non-compositional. For instance, you can't put an image control inside a tab control header. Customizing list views and combo boxes is time-consuming and painful.
- Dynamic layout is difficult to get right reliably

Xamarin

Xamarin, now owned by Microsoft, lets you write mobile apps in C# that target iOS and Android, as well as Windows Phone. Being cross-platform, this runs not on the .NET Framework, but its own framework (a derivation of the open source Mono framework).

UWP (Universal Windows Platform)

UWP is for writing apps that target Windows 10 desktop and devices, distributed via the Windows Store. Its rich-client API is designed for writing touch-first user interfaces, and was inspired by WPF and uses XAML for layout. The namespaces are Windows.UI and Windows.UI.Xaml.

Silverlight

Silverlight is also distinct from the .NET Framework, and lets you write a graphical UI that runs in a web browser, much like Macromedia's Flash. With the rise of HTML5, Microsoft has abandoned Silverlight.

Backend Technologies

ADO.NET

ADO.NET is the managed data access API. Although the name is derived from the 1990s-era ADO (ActiveX Data Objects), the technology is completely different. ADO.NET contains two major low-level components:

Provider layer

The provider model defines common classes and interfaces for low-level access to database providers. These interfaces comprise connections, commands, adapters, and readers (forward-only, read-only cursors over a database). The Framework ships with native support for Microsoft SQL Server, and numerous third-party drivers are available for other databases.

DataSet model

A DataSet is a structured cache of data. It resembles a primitive in-memory database, which defines SQL constructs such as tables, rows, columns, relationships, constraints, and views. By programming against a cache of data, you can reduce the number of trips to the server, increasing server scalability and the responsiveness of a rich-client user interface. DataSets are serializable and are designed to be sent across the wire between client and server applications.

Sitting above the provider layer are three APIs that offer the ability to query databases via LINQ:

- Entity Framework (.NET Framework only)
- Entity Framework Core (.NET Framework and .NET Core)
- LINQ to SQL (.NET Framework only)

LINQ to SQL is simpler than Entity Framework, and has historically produced better SQL (although Entity Framework has benefited from numerous updates).
Entity Framework is more flexible in that you can create elaborate mappings between the database and the classes that you query (Entity Data Model), and offers a model that allows third-party support for databases other than SQL Server.

Entity Framework Core (EF Core) is a rewrite of Entity Framework with a simpler design inspired by LINQ to SQL. It abandons the complex Entity Data Model and runs on both .NET Framework and .NET Core.

Windows Workflow (.NET Framework only)

Windows Workflow is a framework for modelling and managing potentially long running business processes. Workflow targets a standard runtime library, providing consistency and interoperability. Workflow also helps reduce coding for dynamically controlled decision-making trees. Windows Workflow is not strictly a backend technology—you can use it anywhere.

Workflow came originally with .NET Framework 3.0, with its types defined in the System.WorkFlow namespace. Workflow was substantially revised in Framework 4.0; the new types live in and under the System.Activities namespace.

COM+ and MSMQ (.NET Framework only)

The Framework allows you to interoperate with COM+ for services such as distributed transactions, via types in the System.EnterpriseServices namespace.

It also supports MSMQ (Microsoft Message Queuing) for asynchronous, one-way messaging through types in System.Messaging.

Distributed System Technologies

Windows Communication Foundation (WCF)

WCF is a sophisticated communications infrastructure introduced in Framework 3.0. WCF is flexible and configurable enough to make both of its predecessors— Remoting and (.ASMX) Web Services—mostly redundant.

WCF, Remoting, and Web Services are all alike in that they implement the following basic model in allowing a client and server application to communicate:

- On the server, you indicate what methods you'd like remote client applications to be able to call.
- On the client, you specify or infer the signatures of the server methods you'd like to call.
- On both the server and the client, you choose a transport and communication protocol (in WCF, this is done through a binding).
- The client establishes a connection to the server.
- The client calls a remote method, which executes transparently on the server.

WCF further decouples the client and server through service contracts and data contracts. Conceptually, the client sends an (XML or binary) message to an end- point on a remote service, rather than directly invoking a remote method. One of the benefits of this decoupling is that clients have no dependency on the .NET plat- form or on any proprietary communication protocols.

For .NET-to-.NET communication, however, WCF offers richer serialization and better tooling than with REST APIs. It's also potentially faster as it's not tied to HTTP and can use binary serialization.

The types for communicating with WCF are in, and below, the System.Service Model namespace.

Web API

Web API runs over ASP.NET/ASP.NET Core and is architecturally similar to Microsoft's MVC API, except that it's designed to expose services and data instead of web pages. Its advantage over WCF is in allowing you to follow popular REST over HTTP conventions, offering easy interoperability with the widest range of platforms.

REST implementations are internally simpler than the SOAP and WS- protocols that WCF relies on for interoperability. REST APIs are also architecturally more elegant for loosely-coupled systems, building on de-facto standards and making excellent use of what HTTP already provides.

Remoting and .ASMX Web Services (.NET Framework only)

Remoting and .ASMX Web Services are WCF's predecessors. Remoting is almost redundant in WCF's wake, and .ASMX Web Services has become entirely redundant.

Remoting is geared toward tightly coupled applications. A typical example is when the client and server are both .NET applications written by the same company (or companies sharing common assemblies). Communication typically involves exchanging potentially complex custom .NET objects that the Remoting infrastructure serializes and deserializes without needing intervention.

The types for Remoting are in or under System.Runtime.Remoting; the types for Web Services are under System.Web.Services.

Scope of .Net Technology:

Over the period of time, many software have evolved along with many new technologies getting introduced. In this race, one which caught the people's interest was .Net. In a very short time this new technology became the boon for the software developer community and now it's been considered as the most growing career option, which clearly indicates that .Net development still rules.

Its growing popularity has made it the first choice of many experienced and fresher and now one can think of having a great career start in this field outside India too. .Net is now part of many international markets like USA, UAE, South Africa, UK and other developing countries and is heading forward with each passing day. With its every new version .Net technologies is evolving at a fast pace and creating amazing job opportunities for the developers.

The availability of RAD in.Net, which means the Rapid Application Development is the reason behind its success. The plus point of learning this technology is that you can develop as many applications as you want for different platforms and environments. You can even use it for building XML web applications and web services that can excellently run on the Internet. .Net is best suited for developing window based applications, web server programs and applications, which are both PC and mobile compatible. It's easy to transfer feature is what makes it the popular choice.

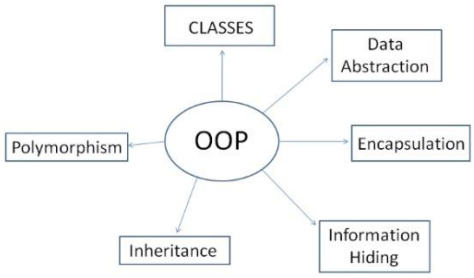
The biggest advantage of learning .net is that one can get a job in various profiles like he/she can be absorbed as a software developer also or a.Net technician too. Today, there are an array of institutes and firms that offer certified and short term course in .Net, which

is a great move from career point of view. Whether you are a diploma holder or an Engineer or an MCA, learning .Net will surely set your career and will offer it a right pace and track. There are ample of career options in this particular field. An interested candidate can go for MCTS(VB.net), MCTS(ASP.net) and MCPD, which are some of the international certifications. You can even choose from Cisco certifications like CCNA, CCNP, CCIE, which will give a new direction to your career.

With so many job prospects in this technology, choosing it will be an ideal choice for your career. This clearly illustrates the future scope of .Net, which is sure to offer you great future ahead in almost all the spheres, ranging from Desktop applications to mobile applications.

Feature of Object Oriented Programming:

Object-oriented programming (OOP) refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure.
In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can inherit characteristics from other objects.



Following are the features of OOPS:
Abstraction: The process of picking out (abstracting) common features of objects and procedures.
Class: A category of objects. The class defines all the common properties of the different objects that belong to it.
Encapsulation: The process of combining elements to create a new entity. A procedure is a type of encapsulation because it combines a series of computer instructions.
Information hiding: The process of hiding details of an object or function. Information hiding is a powerful programming technique because it reduces complexity.
Inheritance: a feature that represents the "is a" relationship between different classes.
Interface: the languages and codes that the applications use to communicate with each other and with the hardware.
Object: a self-contained entity that consists of both data and procedures to manipulate the data.
Polymorphism: A programming language's ability to process objects differently depending on their data type or class.
Procedure: a section of a program that performs a specific task.

Procedure-Oriented vs. Object-Oriented Programming

Object-Oriented Programming (OOP) is a high-level programming language where a program is divided into small chunks called objects using the object-oriented model, hence the name. This paradigm is based on objects and classes.

- **Object –** An object is basically a self-contained entity that accumulates both data and procedures to manipulate the data. Objects are merely instances of classes.
- **Class –** A class, in simple terms, is a blueprint of an object which defines all the common properties of one or more objects that are associated with it. A class can be used to define multiple objects within a program.

The OOP paradigm mainly eyes on the data rather than the algorithm to create modules by dividing a program into data and functions that are bundled within the objects. The modules cannot be modified when a new object is added restricting any non-member function access to the data. Methods are the only way to assess the data.

Objects can communicate with each other through same member functions. This process is known as message passing. This anonymity among the objects is what makes the program secure. A programmer can create a new object from the already existing objects by taking most of its features thus making the program easy to implement and modify.

Procedure-Oriented Programming (POP) follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions. Each step is carried out in order in a systematic manner so that a computer can understand what to do. The program is divided into small parts called functions and then it follows a series of computational steps to be carried out in order.
It follows a top-down approach to actually solve a problem, hence the name. Procedures correspond to functions and each function has its own purpose. Dividing the program into functions is the key to procedural programming. So a number of different functions are written in order to accomplish the tasks.

Difference between OOP and POP is shown below:

Procedure Oriented Programming	Object Oriented Programming
In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.

In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Unit -2 C# Language Basics [12 Hrs]

Writing Console and GUI Applications; Identifiers and keywords; Writing comments; Data Types; Expressions and Operators; Strings and Characters; Arrays; Variables and Parameters; Statements (Declaration, Expression, Selection, Iteration and Jump Statements); Namespaces

A First C# Program

Here is a program that multiplies 12 by 30 and prints the result, 360, to the screen. The double forward slash indicates that the remainder of a line is a comment

```
using System;                   // Importing namespace

class Test                      // Class declaration
{
  static void Main()            //   Method declaratior
  {
    int x = 12 * 30;            //     Statement 1
    Console.WriteLine (x);      //     Statement 2
  }                             //   End of method
}                               // End of class
```

Writing higher-level functions that call upon lower-level functions simplifies a pro- gram. We can refactor our program with a reusable method that multiplies an integer by 12 as follows:

```
using System;

class Test
{
  static void Main()
  {
    Console.WriteLine (FeetToInches (30));      // 360
    Console.WriteLine (FeetToInches (100));     // 1200
  }

  static int FeetToInches (int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

Compilation

The C# compiler compiles source code, specified as a set of files with the .cs extension, into an assembly. An assembly is the unit of packaging and deployment in .NET. An assembly can be either an application or a library. A normal console or Windows application has a Main method and is an .exe file.

A library is a .dll and is equivalent to an .exe without an entry point. Its purpose is to be called upon (referenced) by an application or by other libraries. The .NET Framework is a set of libraries.

The name of the C# compiler is csc.exe. You can either use an IDE such as Visual Studio to compile, or call csc manually from the command line.
To compile manually, first save a program to a file such as MyFirstProgram.cs, and then go to the command line and invoke csc (located in **C:\Windows\Microsoft.NET\Framework\v4.0.30319**) as follows:

csc MyFirstProgram.cs

This produces an application named MyFirstProgram.exe

To produce a library (.dll), do the following:

csc /target:library MyFirstProgram.cs

Identifiers and Keywords

Identifiers are names that programmers choose for their classes, methods, variables, and so on. These are the identifiers in our example program, in the order they appear:

```
System   Test   Main   x   Console   WriteLine
```

An identifier must be a whole word, essentially made up of Unicode characters starting with a letter or underscore. C# identifiers are case-sensitive. By convention, parameters, local variables, and private fields should be in camel case (e.g., myVariable), and all other identifiers should be in Pascal case (e.g., MyMethod).

Keywords are names that mean something special to the compiler. These are the keywords in our example program:

```
using   class   static   void   int
```

Most keywords are reserved, which means that you can't use them as identifiers. Here is the full list of C# reserved keywords (**Total 77**):

```
abstract  do        in          protected   true
as        double    int         public      try
base      else      interface   readonly    typeof
bool      enum      internal    ref         uint
break     event     is          return      ulong
byte      explicit  lock        sbyte       unchecked
case      extern    long        sealed      unsafe
catch     false     namespace   short       ushort
char      finally   new         sizeof      using
checked   fixed     null        stackalloc  virtual
class     float     object      static      void
const     for       operator    string      volatile
continue  foreach   out         struct      while
decimal   goto      override    switch
default   if        params      this
delegate  implicit  private     throw
```

Avoiding conflicts

If you really want to use an identifier that clashes with a reserved keyword, you can do so by qualifying it with the @ prefix. For instance:

```
class class  {...}      // Illegal
class @class {...}      // Legal
```

The @ symbol doesn't form part of the identifier itself. So @myVariable is the same as myVariable.

Contextual keywords

Some keywords are contextual, meaning they can also be used as identifiers— without an @ symbol. These are:

```
add          dynamic  in      orderby  var
ascending    equals   into    partial  when
async        from     join    remove   where
await        get      let     select   yield
by           global   nameof  set
descending   group    on      value
```

Literals, Punctuators, and Operators

Literals are primitive pieces of data lexically embedded into the program. The literals we used in our example program are 12 and 30.

Punctuators help demarcate the structure of the program. These are the punctuators we used in our example program:

```
{   }   ;
```

The **braces** group multiple statements into a statement block.

The **semicolon** terminates a statement. (Statement blocks, however, do not require a semicolon.) Statements can wrap multiple lines:

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

An **operator** transforms and combines expressions. Most operators in C# are denoted with a symbol, such as the multiplication operator, *. We will discuss operators in more detail later in this chapter. These are the operators we used in our example program:

```
.  ()   *   =
```

A **period** denotes a member of something (or a decimal point with numeric literals). **Parentheses** are used when declaring or calling a method; empty parentheses are used when the method accepts no arguments.

An **equals sign** performs assignment. (The double equals sign, ==, performs equality comparison).

Comments

C# offers two different styles of source-code documentation: single-line comments and multiline comments. A single-line comment begins with a double forward slash and continues until the end of the line. For example:

```
int x = 3;   // Comment about assigning 3 to x
```

A multiline comment begins with /* and ends with */. For example:

```
int x = 3;   /* This is a comment that
                spans two lines */
```

Type Basics

A type defines the blueprint for a value. In our example, we used two literals of type int with values 12 and 30. We also declared a variable of type int whose name was x:

```
static void Main()
{
  int x = 12 * 30;
  Console.WriteLine (x);
}
```

A variable denotes a storage location that can contain different values over time. In contrast, a constant always represents the same value.

```
const int y = 360;
```

All values in C# are instances of a type. The meaning of a value, and the set of possible values a variable can have, is determined by its type.

Predefined Type Examples

Predefined types are types that are specially supported by the compiler. The **int type** is a predefined type for representing the set of integers that fit into 32 bits of memory and is the default type for numeric literals within this range.
We can perform functions such as arithmetic with instances of the int type as follows:

```
int x = 12 * 30;
```

Another predefined C# type is **string**. The string type represents a sequence of characters, such as ".NET" or "http://oreilly.com". We can work with strings by calling functions on them as follows:

```
string message = "Hello world";
string upperMessage = message.ToUpper();
Console.WriteLine (upperMessage);              // HELLO WORLD

int x = 2015;
message = message + x.ToString();
Console.WriteLine (message);                   // Hello world2015
```

The predefined bool type has exactly two possible values: true and false. The bool type is commonly used to conditionally branch execution flow based with an if statement. For example:

```
bool simpleVar = false;
if (simpleVar)
  Console.WriteLine ("This will not print");

int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
  Console.WriteLine ("This will print");
```

In C#, predefined types (also referred to as built-in types) are recognized with a C# keyword. The **System namespace** in the .NET Framework contains many important types that are not predefined by C# (e.g., DateTime).

Custom Type Examples

Just as we can build complex functions from simple functions, we can build complex types from primitive types. In this example, we will define a custom type named **UnitConverter** — a class that serves as a blueprint for unit conversions:

```
using System;

public class UnitConverter
{
  int ratio;                                                  // Field
  public UnitConverter (int unitRatio) {ratio = unitRatio; } // Constructor
  public int Convert   (int unit)    {return unit * ratio; } // Method
}

class Test
{
  static void Main()
  {
    UnitConverter feetToInchesConverter = new UnitConverter (12);
    UnitConverter milesToFeetConverter  = new UnitConverter (5280);

    Console.WriteLine (feetToInchesConverter.Convert(30));    // 360
    Console.WriteLine (feetToInchesConverter.Convert(100));   // 1200
    Console.WriteLine (feetToInchesConverter.Convert(
                         milesToFeetConverter.Convert(1)));   // 63360
  }
}
```

A type contains **data members and function members.** The data member of UnitConverter is the field called ratio. The function members of UnitConverter are the Convert method and the UnitConverter's constructor.

Conversions

C# can convert between instances of compatible types. A conversion always creates a new value from an existing one. Conversions can be either **implicit or explicit**: implicit conversions happen automatically, and explicit conversions require a cast.
In the following example, we implicitly convert an int to a long type (which has twice the bitwise capacity of an int) and explicitly cast an int to a short type (which has half the capacity of an int):

```
int x = 12345;       // int is a 32-bit integer
long y = x;          // Implicit conversion to 64-bit integer
short z = (short)x;  // Explicit conversion to 16-bit integer
```

Implicit conversions are allowed when both of the following are true:

- The compiler can guarantee they will always succeed.
- No information is lost in conversion.

Conversely, explicit conversions are required when one of the following is true:

- The compiler cannot guarantee they will always succeed.
- Information may be lost during conversion.

Value Types Versus Reference Types

All C# types fall into the following categories:

- Value types
- Reference types
- Generic type parameters
- Pointer types

Value types comprise most built-in types (specifically, all numeric types, the char type, and the bool type) as well as custom struct and enum types.
Reference types comprise all class, array, delegate, and interface types. (This includes the predefined string type.)
The fundamental difference between value types and reference types is how they are handled in memory.

Value types

The content of a value type variable or constant is simply a value. For example, the content of the built-in value type, int, is 32 bits of data.
You can define a custom value type with the **struct** keyword:

```
public struct Point { public int X; public int Y; }
```

or more tersely:

```
public struct Point { public int X, Y; }
```

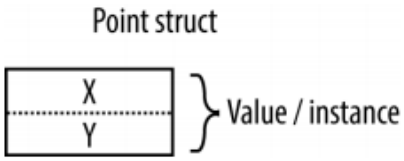


Figure 2-1. A value-type instance in memory

The assignment of a value-type instance always copies the instance. For example:

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1;             // Assignment causes copy

  Console.WriteLine (p1.X);  // 7
  Console.WriteLine (p2.X);  // 7

  p1.X = 9;                  // Change p1.X

  Console.WriteLine (p1.X);  // 9
  Console.WriteLine (p2.X);  // 7
}
```

Figure 2-2 shows that p1 and p2 have independent storage.

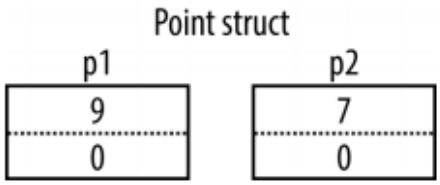


Figure 2-2. Assignment copies a value-type instance

Reference types

A reference type is more complex than a value type, having two parts: an object and the reference to that object. The content of a reference-type variable or constant is a reference to an object that contains the value. Here is the Point type from our previous example rewritten as a class, rather than a struct.

```
public class Point { public int X, Y; }
```

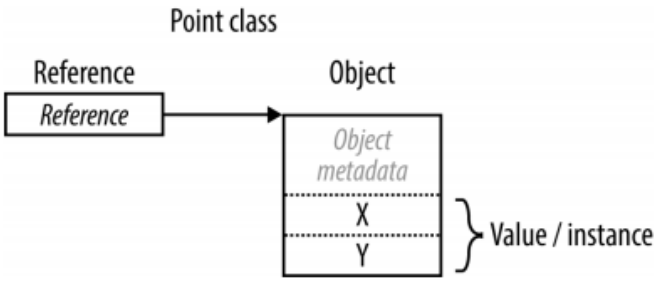


Figure 2-3. A reference-type instance in memory

Numeric Types

C# has the predefined numeric types:

C# type	System type	Suffix	Size	Range
Integral—signed				
sbyte	SByte		8 bits	-2^7 to 2^7-1
short	Int16		16 bits	-2^{15} to $2^{15}-1$
int	Int32		32 bits	-2^{31} to $2^{31}-1$
long	Int64	L	64 bits	-2^{63} to $2^{63}-1$
Integral—unsigned				
byte	Byte		8 bits	0 to 2^8-1
ushort	UInt16		16 bits	0 to $2^{16}-1$
uint	UInt32	U	32 bits	0 to $2^{32}-1$
ulong	UInt64	UL	64 bits	0 to $2^{64}-1$
Real				
float	Single	F	32 bits	$\pm(\sim10^{-45}$ to $10^{38})$
double	Double	D	64 bits	$\pm(\sim10^{-324}$ to $10^{308})$
decimal	Decimal	M	128 bits	$\pm(\sim10^{-28}$ to $10^{28})$

Numeric Conversions

Converting between integral types

Integral type conversions are implicit when the destination type can represent every possible value of the source type. Otherwise, an explicit conversion is required. For example:

```
int x = 12345;       // int is a 32-bit integer
long y = x;          // Implicit conversion to 64-bit integral type
short z = (short)x;  // Explicit conversion to 16-bit integral type
```

Converting between floating-point types

A float can be implicitly converted to a double, since a double can represent every possible value of a float. The reverse conversion must be explicit.

Converting between floating-point and integral types

All integral types may be implicitly converted to all floating-point types:

```
int i = 1;
float f = i;
```

The reverse conversion must be explicit:

```
int i2 = (int)f;
```

When you cast from a floating-point number to an integral type, any fractional portion is truncated; no rounding is per- formed. The static class System.Convert provides methods that round while converting between various numeric types.
Implicitly converting a large integral type to a floating-point type preserves magnitude but may occasionally lose precision. This is because floating-point types always have more magnitude than integral types, but may have less precision. Rewriting our example with a larger number demonstrates this:

```
int i1 = 100000001;
float f = i1;          // Magnitude preserved, precision lost
int i2 = (int)f;       // 100000000
```

Decimal conversions

All integral types can be implicitly converted to the decimal type, since a decimal can represent every possible C# integral-type value. All other numeric conversions to and from a decimal type must be explicit.

Operators in C #:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Miscellaneous Operators

Arithmetic Operators

Following table shows all the arithmetic operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B = 30
-	Subtracts second operand from the first	A - B = -10
*	Multiplies both operands	A * B = 200
/	Divides numerator by de-numerator	B / A = 2
%	Modulus Operator and remainder of after an integer division	B % A = 0
++	Increment operator increases integer value by one	A++ = 11
--	Decrement operator decreases integer value by one	A-- = 9

Relational Operators

Following table shows all the relational operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C#. Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.

!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit by bit operation. The Bitwise operators supported by C# are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = 61, which is 1100 0011 in 2's complement due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15, which is 0000 1111

Assignment Operators

There are following assignment operators supported by C#:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B assigns value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2

&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Miscellaneous Operators

There are few other important operators including **sizeof, typeof** and **? :**supported by C#.

Operator	**Description**	**Example**
sizeof()	Returns the size of a data type.	sizeof(int), returns 4.
typeof()	Returns the type of a class.	typeof(StreamReader);
&	Returns the address of an variable.	&a; returns actual address of the variable.
*	Pointer to a variable.	*a; creates pointer named 'a' to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y
is	Determines whether an object is of a certain type.	If(Ford is Car) // checks if Ford is an object of the Car class.
as	Cast without raising an exception if the cast fails.	Object obj = new StringReader("Hello"); StringReader r = obj as StringReader;

Conditional operator (ternary operator)

The conditional operator (more commonly called the ternary operator, as it's the only operator that takes three operands) has the form q ? a : b, where if condition q is true, a is evaluated, else b is evaluated. For example:

```
static int Max (int a, int b)
{
  return (a > b) ? a : b;
}
```

The conditional operator is particularly useful in LINQ queries

Strings and Characters

C#'s char type (aliasing the System.Char type) represents a Unicode character and occupies 2 bytes. A char literal is specified inside single quotes:

```
char c = 'A';       // Simple character
```

Escape sequences express characters that cannot be expressed or interpreted literally. An escape sequence is a backslash followed by a character with a special meaning. For example:

```
char newLine = '\n';
char backSlash = '\\';
```

The escape sequence characters are shown below.

Char	Meaning	Value
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

The \u (or \x) escape sequence lets you specify any Unicode character via its four-digit hexadecimal code:

```
char copyrightSymbol = '\u00A9';
char omegaSymbol     = '\u03A9';
char newLine         = '\u000A';
```

Char Conversions

An implicit conversion from a char to a numeric type works for the numeric types that can accommodate an unsigned short. For other numeric types, an explicit conversion is required.

String Type

C#'s string type (aliasing the **System.String** type) represents an immutable sequence of Unicode characters. A string literal is specified inside double quotes:

```
string a = "Heat";
```

string is a reference type, rather than a value type. Its equality operators, however, follow value-type semantics:

```
string a = "test";
string b = "test";
Console.Write (a == b);  // True
```

The escape sequences that are valid for char literals also work inside strings:

```
string a = "Here's a tab:\t";
```

The cost of this is that whenever you need a literal backslash, you must write it twice:

```
string a1 = "\\\\server\\fileshare\\helloworld.cs";
```

To avoid this problem, C# allows **verbatim** string literals. A verbatim string literal is prefixed with @ and does not support escape sequences. The following verbatim string is identical to the preceding one:

```
string a2 = @"\\server\fileshare\helloworld.cs";
```

A verbatim string literal can also span multiple lines:

```
string escaped  = "First Line\r\nSecond Line";
string verbatim = @"First Line
Second Line";

// True if your IDE uses CR-LF line separators:
Console.WriteLine (escaped == verbatim);
```

You can include the double-quote character in a verbatim literal by writing it twice:

```
string xml = @"<customer id=""123""></customer>";
```

String concatenation

The + operator concatenates two strings:

```
string s = "a" + "b";
```

One of the operands may be a non-string value, in which case ToString is called on that value. For example:

```
string s = "a" + 5;  // a5
```

Using the + operator repeatedly to build up a string is inefficient: a better solution is to use the **System.Text.StringBuilder** type.

String interpolation

A string preceded with the **$** character is called an **interpolated** string. Interpolated strings can include expressions inside braces:

```
int x = 4;
Console.Write ($"A square has {x} sides");  // Prints: A square has 4 sides
```

String comparisons

string does not support < and > operators for comparisons. You must use the string's **CompareTo** method.

Arrays

An array represents a fixed number of variables (called elements) of a particular type. The elements in an array are always stored in a contiguous block of memory, providing highly efficient access.

An array is denoted with square brackets after the element type. For example:

```
char[] vowels = new char[5];    // Declare an array of 5 characters
```

Square brackets also **index** the array, accessing a particular element by position:

```
vowels[0] = 'a';
vowels[1] = 'e';
vowels[2] = 'i';
vowels[3] = 'o';
vowels[4] = 'u';
Console.WriteLine (vowels[1]);      // e
```

This prints "e" because array indexes start at 0. We can use a for loop statement to iterate through each element in the array. The for loop in this example cycles the integer i from 0 to 4:

```
for (int i = 0; i < vowels.Length; i++)
  Console.Write (vowels[i]);            // aeiou
```

An array initialization expression lets you declare and populate an array in a single step:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

or simply:

```
char[] vowels = {'a','e','i','o','u'};
```

Multidimensional Arrays

Multidimensional arrays come in two varieties**: rectangular and jagged**.

Rectangular arrays represent an n-dimensional block of memory, and jagged arrays are arrays of arrays.

Rectangular arrays

Rectangular arrays are declared using commas to separate each dimension. The following declares a rectangular two-dimensional array, where the dimensions are 3 by 3:

```
int[,] matrix = new int[3,3];
```

A rectangular array can be initialized as follows (to create an array identical to the previous example):

```
int[,] matrix = new int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
```

Example Program,

```
class Rectangular
{
    static void Main(string[] args)
    {
        int[,] vals = new int[4, 2] {
            { 9, 99 },
            { 3, 33 },
            { 4, 44 },
            { 1, 11 }
        };

        for (int i = 0; i < 4; i++)
        {
            for (int j = 0; j < 2; j++)
            {
                Console.WriteLine(vals[i,j]);
            }
        }
            /* Using for each loop
            foreach (var val in vals)
            {
                Console.WriteLine(val);
            }*/
            Console.ReadKey();
    }
}
```

Jagged arrays

Jagged arrays are declared using successive square brackets to represent each dimension. Here is an example of declaring a jagged two-dimensional array, where the outermost dimension is 3:

```
int[][] matrix = new int[3][];
```

The inner dimensions aren't specified in the declaration because, unlike a rectangular array, each inner array can be an arbitrary length. Each inner array is implicitly initialized to null rather than an empty array.

A jagged array can be initialized as follows (to create an array identical to the previous example with an additional element at the end):

```
int[][] matrix = new int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8,9}
};
```

Example Program,

```
class Jagged
{
    static void Main(string[] args)
    {
        int[][] jagged = new int[][]
        {
            new int[] { 1, 2 },
            new int[] { 1, 2, 3 },
            new int[] { 1, 2, 3, 4 }
        };

        foreach (int[] array in jagged)
        {
            foreach (int e in array)
            {
                Console.Write(e + " ");
            }
            Console.Write('\n');
        }

        Console.ReadKey();
    }
}
```

Simplified Array Initialization Expressions

```
char[] vowels = {'a','e','i','o','u'};

int[,] rectangularMatrix =
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};

int[][] jaggedMatrix =
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```

Bounds Checking

All array indexing is bounds-checked by the runtime. An **IndexOutOfRangeException** is thrown if you use an invalid index:

```
int[] arr = new int[3];
arr[3] = 1;               // IndexOutOfRangeException thrown
```

As with Java, array bounds checking is necessary for type safety and simplifies debugging. Generally, the performance hit from bounds checking is minor, and the JIT (Just-In-Time) compiler can perform optimizations, such as determining in advance whether all indexes will be safe before entering a loop, thus avoiding a check on each iteration. In addition, C# provides "unsafe" code that can explicitly bypass bounds checking.

Variables and Parameters

A variable represents a storage location that has a modifiable value. A variable can be a local variable, parameter (value, ref, or out), field (instance or static), or array element.

The Stack and the Heap

The stack and the heap are the places where variables and constants reside. Each has very different lifetime semantics.

Stack

The stack is a block of memory for storing local variables and parameters. The stack logically grows and shrinks as a function is entered and exited. Consider the following method:

```
static int Factorial (int x)
{
  if (x == 0) return 1;
  return x * Factorial (x-1);
}
```

This method is recursive, meaning that it calls itself. Each time the method is entered, a new int is allocated on the stack, and each time the method exits, the int is deallocated.

Heap

The heap is a block of memory in which objects (i.e., reference-type instances) reside. Whenever a new object is created, it is allocated on the heap, and a reference to that object is returned. During a program's execution, the heap starts filling up as new objects are created. The runtime has a garbage collector that periodically deallocates objects from the heap, so your program does not run out of memory. An object is eligible for deallocation as soon as it's not referenced by anything that's itself "alive."

```
using System;
using System.Text;

class Test
{
  static void Main()
  {
    StringBuilder ref1 = new StringBuilder ("object1");
    Console.WriteLine (ref1);
    // The StringBuilder referenced by ref1 is now eligible for GC.

    StringBuilder ref2 = new StringBuilder ("object2");
    StringBuilder ref3 = ref2;
    // The StringBuilder referenced by ref2 is NOT yet eligible for GC.

    Console.WriteLine (ref3);                   // object2
  }
}
```

Value-type instances (and object references) live wherever the variable was declared. If the instance was declared as a field within a class type, or as an array element, that instance lives on the heap.

Definite Assignment

C# enforces a definite assignment policy. In practice, this means that outside of an unsafe context, it's impossible to access uninitialized memory. Definite assignment has three implications:

- Local variables must be assigned a value before they can be read.
- Function arguments must be supplied when a method is called (unless marked as optional.
- All other variables (such as fields and array elements) are automatically initialized by the runtime.

For example, the following code results in a compile-time error:

```
static void Main()
{
  int x;
  Console.WriteLine (x);        // Compile-time error
}
```

Fields and array elements are automatically initialized with the default values for their type. The following code outputs 0, because array elements are implicitly assigned to their default values:

```
static void Main()
{
  int[] ints = new int[2];
  Console.WriteLine (ints[0]);    // 0
}
```

The following code outputs 0, because fields are implicitly assigned a default value:

```
class Test
{
  static int x;
  static void Main() { Console.WriteLine (x); }   // 0
}
```

Default Values

All type instances have a default value. The default value for the predefined types is the result of a bitwise zeroing of memory:

Type	Default value
All reference types	`null`
All numeric and enum types	`0`
`char` type	`'\0'`
`bool` type	`false`

You can obtain the default value for any type with the default keyword,

```
decimal d = default (decimal);
```

Parameters

A method has a sequence of parameters. Parameters define the set of arguments that must be provided for that method. In this example, the method **Foo** has a single parameter named **p**, of type **int**:

```
static void Foo (int p)
{
  p = p + 1;                 // Increment p by 1
  Console.WriteLine (p);     // Write p to screen
}

static void Main()
{
  Foo (8);                  // Call Foo with an argument of 8
}
```

You can control how parameters are passed with the ref and out modifiers:

Parameter modifier	Passed by	Variable must be definitely assigned
(None)	Value	Going *in*
`ref`	Reference	Going *in*
`out`	Reference	Going *out*

Passing arguments by value

By default, arguments in C# are passed by value, which is by far the most common case. This means a copy of the value is created when passed to the method:

```
class Test
{
  static void Foo (int p)
  {
    p = p + 1;                // Increment p by 1
    Console.WriteLine (p);    // Write p to screen
  }

  static void Main()
  {
    int x = 8;
    Foo (x);                  // Make a copy of x
    Console.WriteLine (x);    // x will still be 8
  }
}
```

Assigning **p** a new value does not change the contents of **x**, since **p** and **x** reside in different memory locations.

The ref modifier

To pass by reference, C# provides the **ref** parameter modifier. In the following example, **p and x refer to the same memory locations**:

```
class Test
{
  static void Foo (ref int p)
  {
    p = p + 1;                // Increment p by 1
    Console.WriteLine (p);    // Write p to screen
  }

  static void Main()
  {
    int x = 8;
    Foo (ref  x);             // Ask Foo to deal directly with x
    Console.WriteLine (x);    // x is now 9
  }
}
```

The out modifier

The out modifier is most commonly used to get multiple return values back from a method. For example:

```
class OutParam
{
    static void Pass(int a, int b, out int x, out int y)
    {
        x = a;
        y = b;
    }
    static void main()
    {
        int x, y;
        Pass(10, 20,out x, out y);
        Console.WriteLine(x); //displays 10
        Console.WriteLine(y); //displays 20
    }
}
```

The params modifier

The params parameter modifier may be specified on the last parameter of a method so that the method accepts any number of arguments of a particular type. The parameter type must be declared as an array. For example:

```
class Test
{
  static int Sum (params int[] ints)
  {
    int sum = 0;
    for (int i = 0; i < ints.Length; i++)
      sum += ints[i];                       // Increase sum by ints[i]
    return sum;
  }

  static void Main()
  {
    int total = Sum (1, 2, 3, 4);
    Console.WriteLine (total);              // 10
  }
}
```

Optional parameters

A parameter is optional if it specifies a default value in its declaration:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

Optional parameters may be omitted when calling the method:

```
Foo();      // 23
```

Operator Precedence and Associativity

When an expression contains multiple operators, precedence and associativity determine the order of their evaluation. Operators with higher precedence execute before operators of lower precedence. If the operators have the same precedence, the operator's associativity determines the order of evaluation.

Precedence

The following expression:

```
1 + 2 * 3
```

is evaluated as follows because * has a higher precedence than +:

```
1 + (2 * 3)
```

Left-associative operators

Binary operators (except for assignment, lambda, and null coalescing operators) are left-associative; in other words, they are evaluated from left to right. For example, the following expression:

```
8 / 4 / 2
```

is evaluated as follows due to left associativity:

```
( 8 / 4 ) / 2     // 1
```

You can insert parentheses to change the actual order of evaluation:

```
8 / ( 4 / 2 )     // 4
```

Right-associative operators

The assignment operators, lambda, null coalescing, and conditional operator are right-associative; in other words, they are evaluated from right to left. Right associativity allows multiple assignments such as the following to compile:

```
x = y = 3;
```

This first assigns 3 to y, and then assigns the result of that expression (3) to x.

Null Operators

C# provides two operators to make it easier to work with nulls: the null coalescing operator and the null-conditional operator.

Null Coalescing Operator

The ?? operator is the null coalescing operator. It says "If the operand is non-null, give it to me; otherwise, give me a default value." For example:

```
string s1 = null;
string s2 = s1 ?? "nothing";   // s2 evaluates to "nothing"
```

If the left-hand expression is non-null, the right-hand expression is never evaluated.

Null-conditional Operator

The ?. operator is the null-conditional or "Elvis" operator. It allows you to call methods and access members just like the standard dot operator, except that if the operand on the left is null, the expression evaluates to null instead of throwing a **NullReferenceException**:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString();  // No error; s instead evaluates to null
```

Statements

Functions comprise statements that execute sequentially in the textual order in which they appear. A statement block is a series of statements appearing between braces (the {} tokens).

Declaration Statements

A declaration statement declares a new variable, optionally initializing the variable with an expression. A declaration statement ends in a semicolon. You may declare multiple variables of the same type in a comma-separated list. For example:

```
string someWord = "rosebud";
int someNumber = 42;
bool rich = true, famous = false;
```

A constant declaration is like a variable declaration, except that it cannot be changed after it has been declared, and the initialization must occur with the declaration.

```
const double c = 2.99792458E08;
c += 10;                          // Compile-time Error
```

Local variables

The scope of a local variable or local constant extends throughout the current block. You cannot declare another local variable with the same name in the current block or in any nested blocks. For example:

```
static void Main()
{
  int x;
  {
    int y;
    int x;            // Error - x already defined
  }
  {
    int y;            // OK - y not in scope
  }
  Console.Write (y);  // Error - y is out of scope
}
```

Expression Statements

Expression statements are expressions that are also valid statements. An expression statement must either change state or call something that might change state.
Changing state essentially means changing a variable. The possible expression statements are:

- Assignment expressions (including increment and decrement expressions)
- Method call expressions (both void and non-void)
- Object instantiation expressions

Here are some examples:

```
// Declare variables with declaration statements:
string s;
int x, y;
System.Text.StringBuilder sb;

// Expression statements
x = 1 + 2;                // Assignment expression
x++;                      // Increment expression
y = Math.Max (x, 5);      // Assignment expression
Console.WriteLine (y);    // Method call expression
sb = new StringBuilder(); // Assignment expression
new StringBuilder();      // Object instantiation expression
```

Note: Please change the syntax of the program.

Conditional Operator (? :)

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as –

```
variable x = (expression) ? value if true : value if false
```

Following is an example –

```
public class Test {

   public static void main(String args[]) {
      int a, b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );

      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
}
```

Output

```
Value of b is : 30
Value of b is : 20
```

Control Statements

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. C# program control statements can be put into the following categories: selection, iteration, and jump.

- **Selection statements** allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- **Iteration statements** enable program execution to repeat one or more statements (that is, iteration statements form loops).
- **Jump statements** allow your program to execute in a nonlinear fashion.

C#'s Selection Statements

C# supports two selection statements: if and switch. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

if

The if statement is C#'s conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

if (condition) statement1;
else statement2;

The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

Here, if a is less than b, then a is set to zero. Otherwise, b is set to zero. In no case are they both set to zero.

Nested ifs

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example:

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d; // this if is
  else a = c;        // associated with this else
}
else a = d;          // this else refers to if(i == 10)
```

As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)** because it is the closest **if** within the same block.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder. It looks like this:

```
if(condition)
  statement;
else if(condition)
  statement;
else if(condition)
  statement;
.
.
.
else
  statement;
```

Here is a program that uses an if-else-if ladder to determine which season a particular month is in.

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

Here is the output produced by the program:

```
April is in the Spring.
```

switch

The switch statement is C#'s multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

```
switch (expression) {
  case value1:
    // statement sequence
    break;
  case value2:
    // statement sequence
    break;
.
.
.
  case valueN:

    // statement sequence
    break;
  default:
    // default statement sequence
}
```

Here is a simple example that uses a switch statement:

```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String args[]) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
        case 1:
          System.out.println("i is one.");
          break;
        case 2:
          System.out.println("i is two.");
          break;
        case 3:
          System.out.println("i is three.");
          break;
        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

Iteration Statements

C#'s iteration statements are for, while, do-while and for-each loop. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. A loop statement allows us to execute a statement or group of statements multiple times.

for loop

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times. A for loop is useful when you know how many times a task is to be repeated.
The syntax of a for loop is –

```
for(initialization; Boolean_expression; update) {
   // Statements
}
```

Following is an example code of the for loop in Java.

```
public class Test {

   public static void main(String args[]) {

      for(int x = 10; x < 20; x = x + 1) {
         System.out.print("value of x : " + x );
         System.out.print("\n");
      }
   }
}
```

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

while Loop

A while loop statement in C# programming language repeatedly executes a target statement as long as a given condition is true.
The syntax of a while loop is –

```
while(Boolean_expression) {
   // Statements
}
```

Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
public class Test {

   public static void main(String args[]) {
      int x = 10;

      while( x < 20 ) {
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }
   }
}
```

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

do while loop

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Following is the syntax of a do...while loop –

```
do {
   // Statements
}while(Boolean_expression);
```

Example

```
public class Test {

   public static void main(String args[]) {
      int x = 10;

      do {
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }while( x < 20 );
   }
}
```

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

foreach Loop

The foreach statement iterates over each element in an enumerable object. Most of the types in C# and the .NET Framework that represent a set or list of elements are enumerable. For example, both an array and a string are enumerable. Here is an example of enumerating over the characters in a string, from the first character through to the last:

```
foreach (char c in "beer")   // c is the iteration variable
  Console.WriteLine (c);

OUTPUT:
b
e
e
r
```

It can be used for retrieving elements of array. It is shown below:

```
class ForEach
{
    static void Main(string[] args)
    {
        int[] a={10,20,30,40,50};
        foreach (int val in a)
        {
            Console.WriteLine(val);
        }
        Console.ReadKey();

    }
}
```

Output:

10
20
30
40
50

Nested Loops

Like all other programming languages, C# allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests for loops:

```
// Loops may be nested.
class Nested {
  public static void main(String args[]) {
    int i, j;

    for(i=0; i<10; i++) {
      for(j=i; j<10; j++)
        System.out.print(".");
      System.out.println();
    }
  }
}
```

The output produced by this program is shown here:

```
..........
.........
........
.......
......
.....
....
...
..
.
```

Jump Statements

The C# jump statements are **break, continue, goto, return, and throw**. These statements transfer control to another part of your program.

Using break

In C#, the break statement has three uses. First, as you have seen, it terminates a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto.

Using break to Exit a Loop

By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

```
// Using break to exit a loop.
class BreakLoop {
  public static void main(String args[]) {
    for(int i=0; i<100; i++) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
  }
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

As you can see, although the for loop is designed to run from 0 to 99, the break statement causes it to terminate early, when **i** equals 10.

Using continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action.

In a while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

Here is an example program that uses continue to cause two numbers to be printed on each line:

```
for (int i = 0; i < 10; i++)
{
  if ((i % 2) == 0)       // If i is even,
    continue;             // continue with next iteration

  Console.Write (i + " ");
}

OUTPUT: 1 3 5 7 9
```

Using return

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement.

Example:

```
class A{
    int a,b,sum;
    public int add() {
        a=10;
        b=15;
        sum=a+b;
        return sum;
    }
}

  class B {
    public static void main(String[] args) {
        A obj=new A();
        int res=obj.add();
        System.out.println("Sum of two numbers="+res);
    }
}
```

Output:

Sum of two numbers=25

The goto statement

The goto statement transfers execution to another label within a statement block. The form is as follows:

```
goto statement-label;
```

Or, when used within a switch statement:

```
goto case case-constant;    // (Only works with constants, not patterns)
```

A label is a placeholder in a code block that precedes a statement, denoted with a colon suffix. The following iterates the numbers 1 through 5, mimicking a for loop:

```
int i = 1;
startLoop:
if (i <= 5)
{
  Console.Write (i + " ");
  i++;
  goto startLoop;
}

OUTPUT: 1 2 3 4 5
```

The throw statement

The throw statement throws an exception to indicate an error has occurred.

If(age<18)
throw new ArithmeticExcecption("Not Eligible to Vote");

Namespaces

A namespace is a domain for type names. Types are typically organized into hierarchical namespaces, making them easier to find and avoiding conflicts. For example, the RSA type that handles public key encryption is defined within the following namespace:

```
System.Security.Cryptography
```

The namespace keyword defines a namespace for types within that block. For example:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
  class Class2 {}
}
```

The dots in the namespace indicate a hierarchy of nested namespaces. The code that follows is semantically identical to the preceding example:

```
namespace Outer
{
  namespace Middle
  {
    namespace Inner
    {
      class Class1 {}
      class Class2 {}
    }
  }
}
```

The using Directive

The using directive imports a namespace, allowing you to refer to types without their fully qualified names. The following imports the previous example's Outer.Middle.Inner namespace:

```
using Outer.Middle.Inner;

class Test
{
  static void Main()
  {
    Class1 c;    // Don't need fully qualified name
  }
}
```

using static

From C# 6, you can import not just a namespace, but a specific type, with the using static directive. All static members of that type can then be used without being qualified with the type name. In the following example, we call the Console class's static WriteLine method:

```
using static System.Console;

class Test
{
  static void Main() { WriteLine ("Hello"); }
}
```

Rules Within a Namespace

Name scoping

Names declared in outer namespaces can be used unqualified within inner name- spaces. In this example, Class1 does not need qualification within Inner:

```
namespace Outer
{
  class Class1 {}

  namespace Inner
  {
    class Class2 : Class1  {}
  }
}
```

Name hiding

If the same type name appears in both an inner and an outer namespace, the inner name wins. To refer to the type in the outer namespace, you must qualify its name. For example:

```
namespace Outer
{
  class Foo { }

  namespace Inner
  {
    class Foo { }

    class Test
    {
      Foo f1;          // = Outer.Inner.Foo
      Outer.Foo f2;    // = Outer.Foo
    }
  }
}
```

Repeated namespaces

You can repeat a namespace declaration, as long as the type names within the namespaces don't conflict:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}

namespace Outer.Middle.Inner
{

   class Class2 {}
 }
```

We can even break the example into two source files such that we could compile each class into a different assembly.

Source file 1:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}
```

Source file 2:

```
namespace Outer.Middle.Inner
{
  class Class2 {}
}
```

Nested using directive

You can nest a using directive within a namespace. This allows you to scope the using directive within a namespace declaration. In the following example, Class1 is visible in one scope, but not in another:

```
namespace N1
{
  class Class1 {}
}

namespace N2
{
  using N1;

  class Class2 : Class1 {}
}

namespace N2
{
  class Class3 : Class1 {}   // Compile-time error
}
```

Advanced Namespace Features

Extern

Extern aliases allow your program to reference two types with the same fully qualified name (i.e., the namespace and type name are identical). This is an unusual scenario and can occur only when the two types come from different assemblies. Consider the following example.

Library 1:

```
// csc target:library /out:Widgets1.dll widgetsv1.cs

namespace Widgets
{
  public class Widget {}
}
```

Library 2:

```
// csc target:library /out:Widgets2.dll widgetsv2.cs

namespace Widgets
{
  public class Widget {}
}
```

Application:

```
// csc /r:Widgets1.dll /r:Widgets2.dll application.cs

using Widgets;

class Test
{
  static void Main()
  {
    Widget w = new Widget();
  }
}
```

The application cannot compile, because Widget is ambiguous. Extern aliases can resolve the ambiguity in our application:

```
// csc /r:W1=Widgets1.dll /r:W2=Widgets2.dll application.cs

extern alias W1;
extern alias W2;

class Test
{
  static void Main()
  {
    W1.Widgets.Widget w1 = new W1.Widgets.Widget();
    W2.Widgets.Widget w2 = new W2.Widgets.Widget();

}}
```

Unit -3 Creating Types in C# [12 Hrs]

Classes; Constructors and Destructors; this Reference; Properties; Indexers; Static Constructors and Classes; Finalizers; Dynamic Binding; Operator Overloading; Inheritance; Abstract Classes and Methods; base Keyword; Overloading; Object Type; Structs; Access Modifiers; Interfaces; Enums; Generics

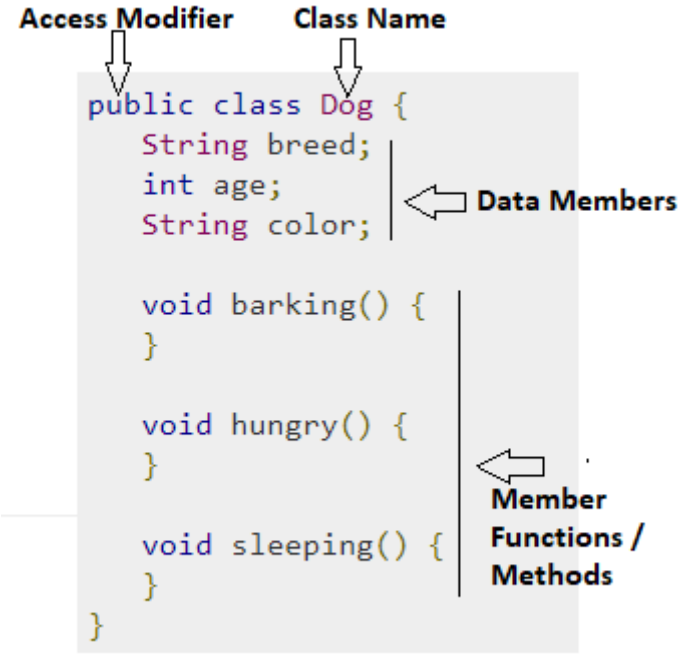
Classes

A class, in the context of C#, are templates that are used to create objects, and to define object data types and methods. Core properties include the data types and methods that may be used by the object. All class objects should have the basic class properties. Classes are categories, and objects are items within each category.

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers** : A class can be public or has default access.
2. **Class name:** The name should begin with a initial letter (capitalized by convention).
3. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the (:). A class can only extend (subclass) one parent.
4. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the (:). A class can implement more than one interface.
5. **Body:** The class body surrounded by braces, { }.

General form of a class is shown below:



Object

An object is a combination of data and procedures working on the available data. An object has a state and behavior. The state of an object is stored in fields (variables), while methods

(functions) display the object's behavior. Objects are created from templates known as classes. In C#, an object is created using the keyword "new". Object is an instance of a class. There are three steps to creating a Java object:

1. Declaration of the object
2. Instantiation of the object
3. Initialization of the object

When a object is declared, a name is associated with that object. The object is instantiated so that memory space can be allocated. Initialization is the process of assigning a proper initial value to this allocated space. The properties of Java objects include:

- One can only interact with the object through its methods. Hence, internal details are hidden.
- When coding, an existing object may be reused.
- When a program's operation is hindered by a particular object, that object can be easily removed and replaced.

A new object t from the class "tree" is created using the following syntax:

Tree t = new Tree ().

Fields

A field is a variable that is a member of a class or struct. For example:

```
class Octopus
{
    string name;
    public int Age = 10;
}
```

Fields allow the following modifiers:

Static modifier	`static`
Access modifiers	`public internal private protected`
Inheritance modifier	`new`
Unsafe code modifier	`unsafe`
Read-only modifier	`readonly`
Threading modifier	`volatile`

The readonly modifier

The readonly modifier prevents a field from being modified after construction. A read-only field can be assigned only in its declaration or within the enclosing type's constructor.

```
public int Age = 10;
static readonly int legs = 8, eyes = 2;
```

Methods

A method performs an action in a series of statements. A method can receive input data from the caller by specifying parameters and output data back to the caller by specifying a return type. A method can specify a void return type, indicating that it doesn't return any value to its caller.

A method can also output data back to the caller via ref/out parameters. A method's signature must be unique within the type. A method's signature com- prises its name and parameter types in order (but not the parameter names, nor the return type).

Methods allow the following modifiers:

Static modifier	`static`
Access modifiers	`public internal private protected`
Inheritance modifiers	`new virtual abstract override sealed`
Partial method modifier	`partial`
Unmanaged code modifiers	`unsafe extern`
Asynchronous code modifier	`async`

Expression-bodied methods

A method that comprises a single expression, such as the following:

```
int Foo (int x) { return x * 2; }
```

can be written more tersely as an expression-bodied method. A fat arrow replaces the braces and return keyword:

```
int Foo (int x) => x * 2;
```

Expression-bodied functions can also have a void return type:

```
void Foo (int x) => Console.WriteLine (x);
```

Overloading methods

A type may overload methods (have multiple methods with the same name), as long as the signatures are different. For example, the following methods can all coexist in the same type:

```
void Foo (int x) {...}
void Foo (double x) {...}
void Foo (int x, float y) {...}
void Foo (float x, int y) {...}
```

However, the following pairs of methods cannot coexist in the same type, since the return type and the params modifier are not part of a method's signature:

```
void  Foo (int x) {...}
float Foo (int x) {...}          // Compile-time error

void  Goo (int[] x) {...}
void  Goo (params int[] x) {...}  // Compile-time error
```

Constructors

A *constructor* in C# is a block of code similar to a method that's called when an instance of an object is created. Here are the key differences between a constructor and a method:

- A constructor doesn't have a return type.
- The name of the constructor must be the same as the name of the class.
- Unlike methods, constructors are not considered members of a class.
- A constructor is called automatically when a new instance of an object is created.

All classes have constructors, whether you define one or not, because C# automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Default Constructor

The *default constructor* is a constructor that is automatically generated in the absence of explicit constructors (i.e. no user defined constructor). The automatically provided constructor is called sometimes a *nullary* constructor.

Following is the syntax of a default constructor –

```
class ClassName {
   ClassName() {
   }
}
```

Instance Constructors

Constructors run initialization code on a class or struct. A constructor is defined like a method, except that the method name and return type are reduced to the name of the enclosing type:

```
public class Panda
{
  string name;                  // Define field
  public Panda (string n)       // Define constructor
  {
    name = n;                   // Initialization code (set up field)
  }
}
...

Panda p = new Panda ("Petey");  // Call constructor
```

Instance constructors allow the following modifiers:

public internal private protected

Overloaded Constructors

A class or struct may overload constructors. To avoid code duplication, one constructor may call another, using **this** keyword:

```
using System;

public class Wine
{
  public decimal Price;
  public int Year;
  public Wine (decimal price) { Price = price; }
  public Wine (decimal price, int year) : this (price) { Year = year; }
}
```

When one constructor calls another, the called constructor executes first.

Destructor

Destructors in C# are methods inside the class used to destroy instances of that class when they are no longer needed. The Destructor is called implicitly by the .NET Framework's Garbage collector and therefore programmer has no control as when to invoke the destructor. An instance variable or an object is eligible for destruction when it is no longer reachable.

Important Points:

- A Destructor is unique to its class i.e. there cannot be more than one destructor in a class.
- A Destructor has no return type and has exactly the same name as the class name (Including the same case).
- It is distinguished apart from a constructor because of the *Tilde symbol (~)* prefixed to its name.
- A Destructor does not accept any parameters and modifiers.
- It cannot be defined in Structures. It is only used with classes.
- It cannot be overloaded or inherited.
- It is called when the program exits.
- Internally, Destructor called the Finalize method on the base class of object.

Syntax

```
class Example
{
    // Rest of the class
    // members and methods.

   // Destructor
   ~Example()
    {
        // Your code
    }

}
```

Example:

```
class ConsDes
    {
        //constructor
        public ConsDes(string message)
        {
            Console.WriteLine(message);
        }

        public void test()
        {
            Console.WriteLine("This is a method");
        }

        //destructor
        ~ConsDes()
        {
            Console.WriteLine("This is a destructor");
            Console.ReadKey();
        }
    }

    class Construct
    {
        static void Main(string[] args)
        {
            string msg = "This is a constructor";
            ConsDes obj = new ConsDes(msg);
            obj.test();
        }
    }
```

Output:

```
This is a constructor
This is a method
This is a destructor
```

Static Constructor

In c#, **Static Constructor** is used to perform a particular action only once throughout the application. If we declare a constructor as **static**, then it will be invoked only once irrespective of number of class instances and it will be called automatically before the first instance is created.

Generally, in c# the static constructor will not accept any access modifiers and parameters. In simple words we can say it's a parameter less.

Following are the properties of static constructor in c# programming language.

- Static constructor in c# won't accept any parameters and access modifiers.

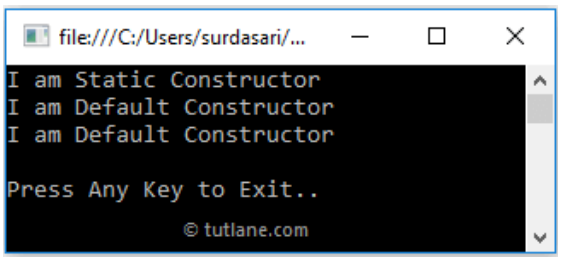
- The static constructor will invoke automatically, whenever we create a first instance of class.
- The static constructor will be invoked by CLR so we don't have a control on static constructor execution order in c#.
- In c#, only one static constructor is allowed to create.

C# Static Constructor Syntax

```
class User
{
   // Static Constructor
   static User()
   {
      // Your Custom Code
   }
}
```

Example

```
class User
    {
        // Static Constructor
        static User()
        {
            Console.WriteLine("I am Static Constructor");
        }
        // Default Constructor
        public User()
        {
            Console.WriteLine("I am Default Constructor");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // Both Static and Default constructors will invoke for
first instance
            User user = new User();
            // Only Default constructor will invoke
            User user1 = new User();
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
```



The this Reference

The “this” keyword in C# is used to refer to the current instance of the class. It is also used to differentiate between the method parameters and class fields if they both have the same name.
Another usage of “this” keyword is to call another constructor from a constructor in the same class.
Here, for an example, we are showing a record of Students i.e: id, Name, Age, and Subject. To refer to the fields of the current class, we have used the “this” keyword in C#:

```
public Student(int id, String name, int age, String subject) {
   this.id = id;
   this.name = name;
   this.subject = subject;
   this.age = age;
}
```

Let us see the complete example to learn how to work with the “this” keyword in C#:

```
using System;

class Student {
   public int id, age;
   public String name, subject;

   public Student(int id, String name, int age, String subject) {
      this.id = id;
      this.name = name;
      this.subject = subject;
      this.age = age;
   }

   public void showInfo() {
      Console.WriteLine(id + " " + name+" "+age+ " "+subject);
   }
}

class StudentDetails {
   public static void Main(string[] args) {
      Student std1 = new Student(001, "Jack", 23, "Maths");

      std1.showInfo();

   }
}
```

Output:

```
001   Jack    23   Maths
```

Properties

Properties look like fields from the outside, but internally they contain logic, like methods do. Properties are named members of classes, structures, and interfaces. Member variables or methods in a class or structures are called **Fields**. Properties are an extension of fields and are accessed using the same syntax. They use **accessors(get and set)** through which the values of the private fields can be read, written or manipulated.

Usually, inside a class, we declare a data field as private and will provide a set of public SET and GET methods to access the data fields. This is a good programming practice since the data fields are not directly accessible outside the class. We must use the set/get methods to access the data fields.

An example, which uses a set of set/get methods, is shown below.

```
using System;
class MyClass
{
    private int x;
    public void SetX(int i)
    {
        x = i;
    }
    public int GetX()
    {
        return x;
    }
}
class MyClient
{
    public static void Main()
    {
        MyClass mc = new MyClass();
        mc.SetX(10);
        int xVal = mc.GetX();
        Console.WriteLine(xVal);
    }
}
```

Output:

10

Automatic Properties

he most common implementation for a property is a getter and/or setter that sim- ply reads and writes to a private field of the same type as the property. An automatic property declaration instructs the compiler to provide this implementation. We can improve the first example in this section by declaring CurrentPrice as an auto- matic property:

```
public class Stock
{
  ...
  public decimal CurrentPrice { get; set; }
}
```

```
class Chk
    {
        public int a { get;set;}
        public int b { get; set; }
        public int sum
        {
            get { return a + b; }
        }

    }

    class Test
    {
        static void Main()
        {
            Chk obj = new Chk();
            obj.a = 10;
            obj.b = 5;
         Console.WriteLine("Sum of "+obj.a+" and "+obj.b+" = "+obj.sum);
            Console.ReadKey();
        }
    }
```

Output:

```
Sum of 10 and 5 = 15
```

Indexers

Indexers provide a natural syntax for accessing elements in a class or struct that encapsulate a list or dictionary of values. Indexers are similar to properties, but are accessed via an index argument rather than a property name.

The string class has an indexer that lets you access each of its char values via an int index:

```
string s = "hello";
Console.WriteLine (s[0]); // 'h'
Console.WriteLine (s[3]); // 'l'
```

The syntax for using indexers is like that for using arrays, except that the index argument(s) can be of any type(s).

C# indexers are usually known as smart arrays. A C# indexer is a class property that allows you to access a member variable of a class or struct using the features of an array. In C#, indexers are created using this keyword. Indexers in C# are applicable on both classes and structs.

Defining an indexer allows you to create a class like that can allows its items to be accessed an array. Instances of that class can be accessed using the [] array access operator.

```
<modifier> <return type> this [argument list]
{
get
{
// your get block code
}
set
{
// your set block code
}
}
```

In the above code:

<modifier>
can be private, public, protected or internal.

<return type>
can be any valid C# types.

this
this is a special keyword in C# to indicate the object of the current class.

[argument list]
The formal-argument-list specifies the parameters of the indexer.

Following program demonstrates how to use an indexer.

```
class Program
{
    class IndexerClass
    {
        private string[] names = new string[10];
        public string this[int i]
        {
            get
            {
                return names[i];
            }
            set
            {
                names[i] = value;
            }
        }
    }
    static void Main(string[] args)
    {
        IndexerClass Team = new IndexerClass();
        Team[0] = "Rocky";
        Team[1] = "Teena";
        Team[2] = "Ana";
        Team[3] = "Victoria";
        Team[4] = "Yani";
        Team[5] = "Mary";
        Team[6] = "Gomes";
        Team[7] = "Arnold";
        Team[8] = "Mike";
        Team[9] = "Peter";
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(Team[i]);
        }
        Console.ReadKey();
    }
}
```

Difference between Indexers and Properties

Indexers	Properties
Indexers are created with this keyword.	Properties don't require this keyword.
Indexers are identified by signature.	Properties are identified by their names.
Indexers are accessed using indexes.	Properties are accessed by their names.
Indexer are instance member, so can't be static.	Properties can be static as well as instance members.
A get accessor of an indexer has the same formal parameter list as the indexer.	A get accessor of a property has no parameters.
A set accessor of an indexer has the same formal parameter list as the indexer, in addition to the value parameter.	A set accessor of a property contains the implicit value parameter.

Static Classes

A C# static class is a class that can't be instantiated. The sole purpose of the class is to provide blueprints of its inherited classes. A static class is created using the "static" keyword in C#. A static class can contain static members only. You can‘t create an object for the static class.

Advantages of Static Classes

1. If you declare any member as a non-static member, you will get an error.
2. When you try to create an instance to the static class, it again generates a compile time error, because the static members can be accessed directly with its class name.
3. The static keyword is used before the class keyword in a class definition to declare a static class.
4. A static class members are accessed by the class name followed by the member name.

Syntax of static class

```
static class classname
{
   //static data members
   //static methods
}
```

Static members of a class

If we declare any members of a class as static we can access it without creating object of that class.

Example

```
class MyCollege
{
    //static fields
    public static string CollegeName;
    public static string Address;

    //static constructor
    static MyCollege()
    {
        CollegeName = "ABC College of Technology";
        Address = "Hyderabad";
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(MyCollege.CollegeName);
        Console.WriteLine(MyCollege.Address);
        Console.Read();
    }
}
```

Example of static class

```
// Creating static class
// Using static keyword
static class Author {

    // Static data members of Author
    public static string A_name = "Ankita";
    public static string L_name = "CSharp";
    public static int T_no = 84;

    // Static method of Author
    public static void details()
    {
        Console.WriteLine("The details of Author is:");
    }
}

// Driver Class
public class GFG {

    // Main Method
    static public void Main()
    {

        // Calling static method of Author
        Author.details();

        // Accessing the static data members of Author
        Console.WriteLine("Author name : {0} ", Author.A_name);
        Console.WriteLine("Language : {0} ", Author.L_name);
        Console.WriteLine("Total number of articles : {0} ",
                                              Author.T_no);
    }
}
```

Output

The details of Author is:
Author name : Ankita
Language : CSharp
Total number of articles : 84

Finalizers

Finalizers are class-only methods that execute before the garbage collector reclaims the memory for an unreferenced object. The syntax for a finalizer is the name of the class prefixed with the ~ symbol:

```
class Class1
{
  ~Class1()
  {
    ...
  }
}
```

Inheritance

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in java by which one class is allow to **inherit** the features (**fields and methods**) of another class.

The process by which one class acquires the properties (data members) and functionalities(methods) of another class is called **inheritance**. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship. Inheritance is used in java for the following:

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
class Subclass-name : Superclass-name
{
  //methods and fields
}
```

The **:** indicates that you are making a new class that derives from an existing class.

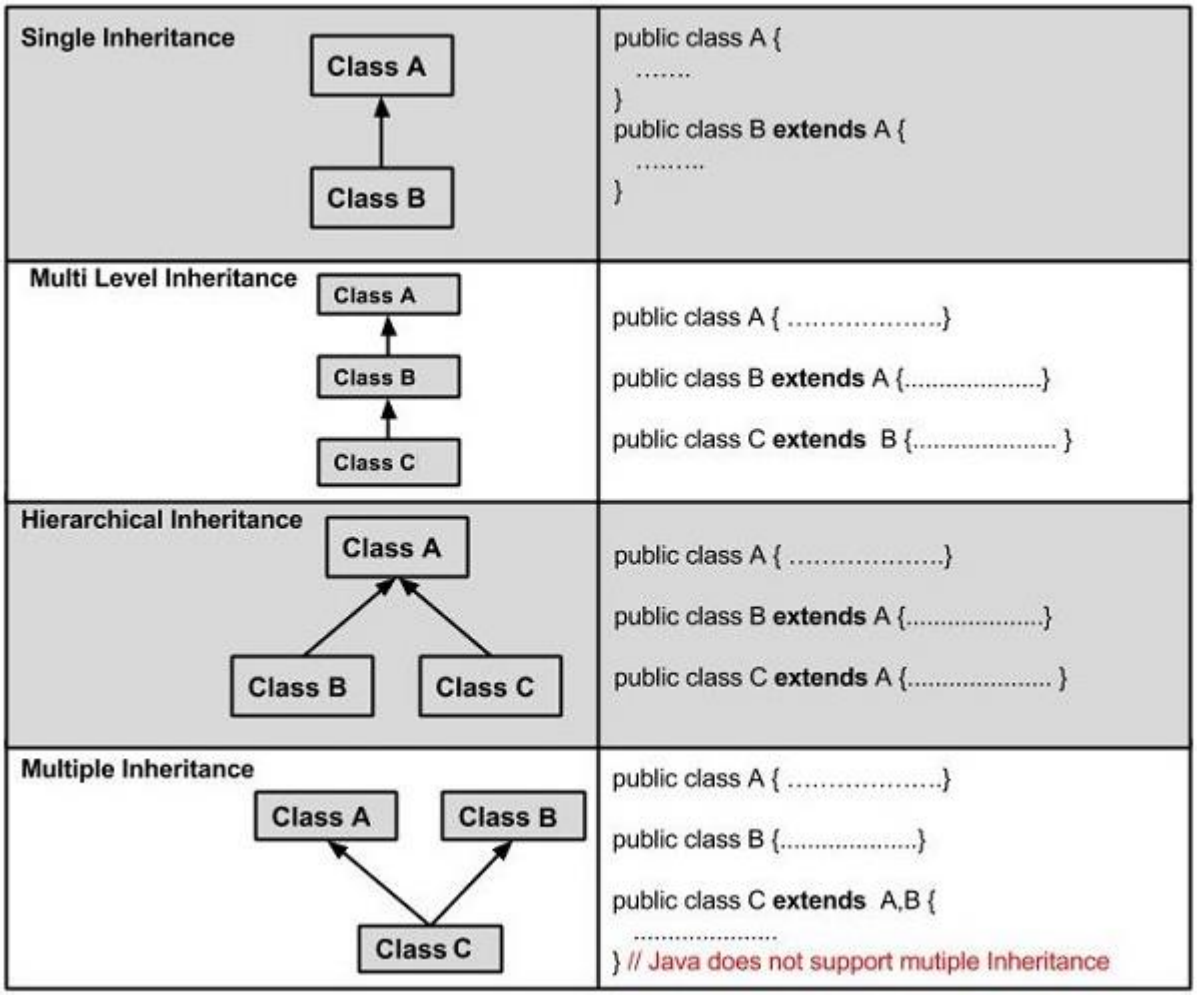
The meaning of "extends" is to increase the functionality.

In the terminology of C#, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Types of inheritance in C#

On the basis of class, there can be **three** types of inheritance in java: **single, multilevel and hierarchical**.

In C# programming, **multiple and hybrid inheritance** is supported through **interface** only.



Note: Codes are written in Java, So please change the syntax of the program according to C# language.

1. Single Inheritance

Single Inheritance refers to a child and parent class relationship where a class extends the another class.

```
class A
    {
        public int a=10, b=5;
    }

    class B : A
    {
        int a=30, b=5;
        public void test()
        {
            Console.WriteLine("Value of a is: "+a);
            Console.WriteLine("Value of a is: " + base.a);
        }
    }
```

```
//driver class
    class Inherit
    {
        static void Main(string[] args)
        {
            B obj = new B();
            obj.test();
            Console.ReadLine();
        }
    }
```

2. Multilevel Inheritance

Multilevel inheritance refers to a child and parent class relationship where a class extends the child class. For example, class C extends class B and class B extends class A.

```
class A5
    {
        public int a, b, c;

        public void ReadData(int a, int b)
        {
            this.a = a;
            this.b = b;
        }

        public void Display()
        {
            Console.WriteLine("Value of a is: "+a);
            Console.WriteLine("Value of b is: " + b);
        }
    }

    class A6 : A5
    {
        public void Add()
        {
            base.c = base.a + base.b;
            Console.WriteLine("Sum="+base.c);
        }
    }

    class A7 : A6
    {
        public void Sub()
        {
            base.c = base.a - base.b;
            Console.WriteLine("Difference=" + base.c);
        }
```

```
}


class Level
{
    static void Main()
    {
        A7 obj = new A7();
        obj.ReadData(20,5);
        obj.Display();
        obj.Add();
        obj.Sub();

        Console.ReadLine();
    }
}
```

3. Hierarchical Inheritance

Hierarchical inheritance refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.

```
class Polygon
    {
        public int dim1,dim2;
        public void ReadDimension(int dim1, int dim2)
        {
            this.dim1 = dim1;
            this.dim2 = dim2;
        }
    }

    class Rectangle : Polygon
    {
        public void AreaRec()
        {
            base.ReadDimension(10,5);
            int area = base.dim1 * base.dim2;
            Console.WriteLine("Area of Rectangle="+area);
        }
    }

    class Traingle : Polygon
    {
        public void AreaTri()
        {
            base.ReadDimension(10,5);
            double area = 0.5*base.dim1 * base.dim2;
            Console.WriteLine("Area of Triangle=" + area);
```

```
        }
    }

    //driver class
    class Hier
    {
        static void Main()
        {
            Traingle tri = new Traingle();
            //tri.ReadDimension(10,5);
            tri.AreaTri();

            Rectangle rec = new Rectangle();
            //rec.ReadDimension(10,7);
            rec.AreaRec();

            Console.ReadLine();
        }
    }
```

4. Multiple Inheritance

When one class extends more than one classes then this is called multiple inheritance. For example: Class C extends class A and B then this type of inheritance is known as multiple inheritance.

C# doesn't allow multiple inheritance. We can use **interfaces** instead of **classes** to achieve the same purpose.

```
interface IA
    {
        // doesn't contain fields
        int CalculateArea();
        int CalculatePerimeter();
    }

    class CA
    {
        public int l, b;
        public void ReadData(int l,int b)
        {
            this.l = l;
            this.b = b;
        }
    }

    class BB : CA, IA
    {
        public int CalculateArea()
        {
            ReadData(10,5);
            int area=l*b;
```

```
            return area;
        }

        public int CalculatePerimeter()
        {
            ReadData(15, 10);
            int peri = 2*(l+b);
            return peri;
        }
    }

    //driver class
    class Inter
    {
        static void Main(string[] args)
        {
            BB obj = new BB();
            //int area=obj.CalculateArea();
            //int peri=obj.CalculatePerimeter();

          Console.WriteLine("Area of Rectangle=" + obj.CalculateArea());
          Console.WriteLine("Perimeter of Rectangle=" +
                  obj.CalculatePerimeter());

            Console.ReadKey();
        }
    }
```

Interface in C#

An interface looks like a class, but has no implementation. The only thing it contains are declarations of events, indexers, methods and/or properties. The reason interfaces only provide declarations is because they are inherited by structs and classes, that must provide an implementation for each interface member declared.

Like a class, an interface can have methods and properties, but the methods declared in interface are by default abstract (only method signature, no body).

- ❖ Interfaces specify what a class must do and not how. It is the blueprint of the class.
- ❖ An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move (). So it specifies a set of methods that the class has to implement.
- ❖ If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.

Why do we use interface?

- It is used to achieve total abstraction.
- Since C# does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.

```
interface IA
    {
        // doesn't contain fields
        void GetData(int l,int b);
        int CalculateArea();
        int CalculatePerimeter();
    }

    class BB : IA
    {
        int l, b;
        public void GetData(int l, int b)
        {
            this.l = l;
            this.b = b;
        }
        public int CalculateArea()
        {
            int area=l*b;
            return area;
        }

        public int CalculatePerimeter()
        {
            int peri = 2*(l+b);
            return peri;
        }
    }

    //driver class
    class Inter
    {
        static void Main(string[] args)
        {
            BB obj = new BB();
            obj.GetData(10,5);
          Console.WriteLine("Area of Rectangle=" + obj.CalculateArea());
          Console.WriteLine("Perimeter of Rectangle=" +
                  obj.CalculatePerimeter());

            Console.ReadKey();
        }
       }
```

Output:

Area of Rectangle=50
Perimeter of Rectangle=30

Abstract Classes

A class declared as abstract can never be instantiated. Instead, only its concrete sub-classes can be instantiated.

If a class is defined as abstract then we can't create an instance of that class. By the creation of the derived class object where an abstract class is inherit from, we can call the method of the abstract class.

For example,

```
abstract class mcn {
    public int add(int a, int b) {
        return (a + b);
    }
}
class mcn1: mcn {
    public int mul(int a, int b) {
        return a * b;
    }
}
class test {
    static void Main(string[] args) {
        mcn1 ob = new mcn1();
        int result = ob.add(5, 10);
        Console.WriteLine("the result is {0}", result);
    }
}
```

Abstract Members

An Abstract method is a method without a body. The implementation of an abstract method is done by a derived class. When the derived class inherits the abstract method from the abstract class, it must override the abstract method. This requirement is enforced at compile time and is also called dynamic polymorphism.

Abstract members are used to achieve total abstraction.

The syntax of using the abstract method is as follows:

<access-modifier>abstract<return-type>method name (parameter)

The abstract method is declared by adding the abstract modifier the method.

```
using System;
public abstract class Shape
{
   public abstract void draw();
}
public class Rectangle : Shape
{
   public override void draw()
   {
      Console.WriteLine("drawing rectangle...");
   } }
```

```
public class TestAbstract
{
   public static void Main()
   {
      Rectangle s = new Rectangle();
      s.draw();
   }
}
```

Output:

```
drawing ractangle...
```

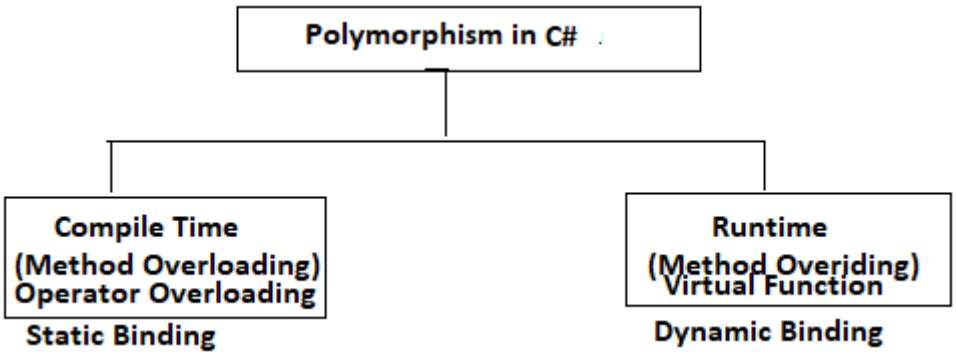
Another Example

```
abstract class test1 {
    public int add(int i, int j) {
        return i + j;
    }
    public abstract int mul(int i, int j);
}
class test2: test1 {
    public override int mul(int i, int j) {
        return i * j;
    }
}
class test3: test1 {
    public override int mul(int i, int j) {
        return i - j;
    }
}
class test4: test2 {
    public override int mul(int i, int j) {
        return i + j;
    }
}
class myclass {
    public static void main(string[] args) {
        test2 ob = new test4();
        int a = ob.mul(2, 4);
        test1 ob1 = new test2();
        int b = ob1.mul(4, 2);
        test1 ob2 = new test3();
        int c = ob2.mul(4, 2);
        Console.Write("{0},{1},{2}", a, b, c);
        Console.ReadLine();
    }
}
```

In the above program, one method i.e. mul can perform various functions depending on the value passed as parameters by creating an object of various classes which inherit other classes. Hence we can acheive dynamic polymorphism with the help of an abstract method.

Polymorphism

Polymorphism in C# is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.



Method Overloading

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists. In order to overload a method, the argument lists of the methods must differ in either of these:

a) Number of parameters.

add(int, int)
add(int, int, int)

b) Data type of parameters.

add(int, int)
add(int, float)

c) Sequence of Data type of parameters.

add(int, float)
add(float, int)

Example of Method Overloading

```
public class Methodoveloading
  {
    public int add(int a, int b)  //two int type Parameters method
    {
        return a + b;

    }
    public int add(int a, int b,int c)  //three int type Parameters with same method
    {
        return a + b+c;

    }
    public float add(float a, float b,float c,float d)  //four float type Parameters
    {
        return a + b+c+d;

    }
  }
```

Method Overriding

Method overriding in C# allows programmers to create base classes that allows its inherited classes to override same name methods when implementing in their class for different purpose. This method is also used to enforce some must implement features in derived classes.

Important points:

- Method overriding is only possible in derived classes, not within the same class where the method is declared.
- Base class must use the virtual or abstract keywords to declare a method. Then only can a method be overridden

Here is an example of method overriding.

```
public class Account
    {
        public virtual int balance()
        {
            return 10;
        }
    }

    public class Amount : Account
    {

        public override int balance()
        {
            return 500;
        }
    }
```

```
class Test
    {
        static void Main()
        {
            Amount obj = new Amount();
            int balance = obj.balance();
            Console.WriteLine("Balance is: "+balance);
            Console.ReadKey();
        }
    }
```

Output:

```
Balance is 500
```

Virtual Method

A virtual method is a method that can be redefined in derived classes. A virtual method has an implementation in a base class as well as derived the class. It is used when a method's basic functionality is the same but sometimes more functionality is needed in the derived class. A virtual method is created in the base class that can be overriden in the derived class. We create a virtual method in the base class using the virtual keyword and that method is overriden in the derived class using the override keyword.

Features of virtual method

- By default, methods are non-virtual. We can't override a non-virtual method.
- We can't use the virtual modifier with the static, abstract, private or override modifiers.
- **If class is not inherited, behaviour of virtual method is same as non-virtual method, but in case of inheritance it is used for method overriding.**

Behaviour of virtual method without inheritance – same as non virtual

```
class Vir
    {
        public virtual void message()
        {
            Console.WriteLine("This is test");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Vir obj = new Vir();
            obj.message();
            Console.ReadKey();
        }
    }
```

Behaviour of virtual method with inheritance – used for overriding

```
class Vir
    {
        public virtual void message()
        {
            Console.WriteLine("This is test");
        }
    }

    class Vir1 : Vir
    {
        public override void message()
        {
            Console.WriteLine("This is test1");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Vir1 obj = new Vir1();
            obj.message();
            Console.ReadKey();
        }
    }
```

Output:

```
This is test1
```

Upcasting and Downcasting

Upcasting converts an object of a specialized type to a more general type. **An upcast operation creates a base class reference from a subclass reference**.

For example:

```
Stock msft = new Stock();
Asset a = msft; // Upcast
```

Downcasting converts an object from a general type to a more specialized type. **A downcast operation creates a subclass reference from a base class reference.**

For example:

```
Stock msft = new Stock();
Asset a = msft; // Upcast
Stock s = (Stock)a; // Downcast
```


```
BankAccount    ba1,
               ba2 =   new BankAccount("John", 250.0M, 0.01);
LotteryAccount la1,
               la2 =   new LotteryAccount("Bent", 100.0M);

ba1 = la2;                  // upcasting   - OK
//  la1 = ba2;              // downcasting - Illegal
                            //   discovered at compile time
//  la1 = (LotteryAccount)ba2;  // downcasting - Illegal
                            //   discovered at run time
la1 = (LotteryAccount)ba1;  // downcasting - OK
                            //   ba1 already refers to a LotteryAccount
```

The as operator

The as operator performs a downcast that evaluates to null (rather than throwing an exception) if the downcast fails:

```
Asset a = new Asset();
Stock s = a as Stock; // s is null; no exception thrown
```

The is operator

The is operator tests whether a reference conversion would succeed; in other words, whether an object derives from a specified class (or implements an inter- face). It is often used to test before downcasting.

```
if (a is Stock)
        Console.WriteLine (((Stock)a).SharesOwned);
```

Operator Overloading

The concept of overloading a function can also be applied to operators. Operator overloading gives the ability to use the same operator to do various operations. It provides additional capabilities to C# operators when they are applied to user-defined data types. It enables to make user-defined implementations of various operations where one or both of the operands are of a user-defined class.

Only the predefined set of C# operators can be overloaded. To make operations on a user-defined data type is not as simple as the operations on a built-in data type. To use operators with user-defined data types, they need to be overloaded according to a programmer's requirement. An operator can be overloaded by defining a function to it. The function of the operator is declared by using the operator keyword.

Syntax:

```
access specifier  className  operator Operator_symbol (parameters)
      {
          // Code
      }
```

The following table describes the overloading ability of the various operators available in C# :

OPERATORS	DESCRIPTION
+, -, !, ~, ++, – –	unary operators take one operand and can be overloaded.
+, -, *, /, %	Binary operators take two operands and can be overloaded.
==, !=, =	Comparison operators can be overloaded.
&&, ||	Conditional logical operators cannot be overloaded directly
+=, -+, *=, /=, %=, =	Assignment operators cannot be overloaded.

Overloading Unary Operators

The following program overloads the **unary - operator** inside the class Complex.

```
using System;
class Complex
{
    private int x;
    private int y;
    public Complex()
    {
    }
    public Complex(int i, int j)
    {
        x = i;
        y = j;
    }
    public void ShowXY()
    {
        Console.WriteLine("{0} {1}", x, y);
    }
    public static Complex operator -(Complex c)
    {
        Complex temp = new Complex();
        temp.x = -c.x;
        temp.y = -c.y;
        return temp;
    }
}
class MyClient
{
    public static void Main()
    {
        Complex c1 = new Complex(10, 20);
        c1.ShowXY(); // displays 10 & 20
        Complex c2 = new Complex();
        c2.ShowXY(); // displays 0 & 0
        c2 = -c1;
        c2.ShowXY(); // diapls -10 & -20
    }
}
```

Overloading Binary Operators

An overloaded binary operator must take two arguments; at least one of them must be of the type class or struct, in which the operation is defined.

```
using System;
class Complex
{
    private int x;
    private int y;
    public Complex()
}
public Complex(int i, int j)
{
    x = i;
    y = j;
}
public void ShowXY()
{
    Console.WriteLine("{0} {1}", x, y);
}
public static Complex operator +(Complex c1, Complex c2)
{
    Complex temp = new Complex();
    temp.x = c1.x + c2.x;
    temp.y = c1.y + c2.y;
    return temp;
}
}
class MyClient
{
    public static void Main()
    {
        Complex c1 = new Complex(10, 20);
        c1.ShowXY(); // displays 10 & 20
        Complex c2 = new Complex(20, 30);
        c2.ShowXY(); // displays 20 & 30
        Complex c3 = new Complex();
        c3 = c1 + c2;
        c3.ShowXY(); // dislplays 30 & 50
    }
}
```

Sealing Functions and Classes

C# Sealed Class

Sealed classes are used to restrict the inheritance feature of object oriented programming. Once a class is defined as a sealed class, this class cannot be inherited. In C#, the sealed modifier is used to declare a class as sealed. If a class is derived from a sealed class, compiler throws an error.

If you have ever noticed, structs are sealed. You cannot derive a class from a struct.

```
// Sealed class
sealed class SealedClass{
}
```

```
using System;
class Class1
{
    static void Main(string[] args)
    {
        SealedClass sealedCls = new SealedClass();
        int total = sealedCls.Add(4, 5);
        Console.WriteLine("Total = " + total.ToString());
    }
}
// Sealed class
sealed class SealedClass
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

Sealed Methods and Properties

You can also use the sealed modifier on a method or a property that overrides a virtual method or property in a base class.

This enables you to allow classes to derive from your class and prevent other developers that are using your classes from overriding specific virtual methods and properties.

```
class X
{
    protected virtual void F()
    {
        Console.WriteLine("X.F");
    }
    protected virtual void F2()
    {
        Console.WriteLine("X.F2");
    }
}
class Y : X
{
    sealed protected override void F()
    {
        Console.WriteLine("Y.F");
    }
    protected override void F2()
    {
        Console.WriteLine("X.F3");
    }
}
class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    //
    protected override void F()
    {
         Console.WriteLine("C.F");
    }
    // Overriding F2 is allowed.
    protected override void F2()
    {
        Console.WriteLine("Z.F2");
    }
}
```

The base Keyword

We can use the base keyword to access the fields of the base class within derived class. It is useful if base and derived classes have the same fields.

If derived class doesn't define same field, there is no need to use base keyword. Base class field can be directly accessed by the derived class.

```
using System;
public class Animal{
   public string color = "white";
}
public class Dog: Animal
{
   string color = "black";
   public void showColor()
   {
      Console.WriteLine(base.color);  //displays white
      Console.WriteLine(color);  //displays black
   }

}
public class TestBase
{
   public static void Main()
   {
      Dog d = new Dog();
      d.showColor();
   }
}
```

The base keyword has two uses:

- To call a base class constructor from a derived class constructor.
- To call a base class method which is overridden in the derived class.

Calling a base class constructor from a derived class constructor

```
class Base
    {
        public Base(int a, int b)
        {
            Console.WriteLine("Value of a={0} and b={1}",a,b);
        }
    }
    class Derived : Base
    {
        public Derived(int x,int y):base(x,y)
        {
```

```
            Console.WriteLine("Value of x={0} and y={1}", x, y);
        }
    }
    class BaseEx
    {
        static void Main(){
            new Derived(10,5);
            Console.ReadKey();
        }
    }
```

Output:

```
Value of a=10 and b=5
Value of x=10 and y=5
```

Calling a base class method which is overridden in derived class

```
class Base
    {
        public virtual void BaseMethod()
        {
            Console.WriteLine("I am inside base class");
        }
    }
    class Derived : Base
    {
        public override void BaseMethod()
        {
            base.BaseMethod();
            Console.WriteLine("I am inside derived class");
        }
    }
    class BaseEx
    {
        static void Main(){
            Derived obj = new Derived();
            obj.BaseMethod();
            Console.ReadKey();
        }
    }
```

Output:

```
I am inside base class
I am inside derived class
```

The object Type

object (System.Object) is the ultimate base class for all types. Any type can be upcast to object.

To illustrate how this is useful, consider a general-purpose stack. A stack is a data structure based on the principle of LIFO—"Last-In First-Out." A stack has two operations: push an object on the stack, and pop an object off the stack.

Here is a simple implementation that can hold up to 10 objects:

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object obj)   { data[position++] = obj;  }
  public object Pop()             { return data[--position]; }
}
```

Because Stack works with the object type, we can Push and Pop instances of *any type* to and from the Stack:

Boxing and Unboxing

Boxing is the act of converting a value-type instance to a reference-type instance. The reference type may be either the object class or an interface.

In this example, we box an int into an object:

```
int x = 9;
object obj = x; // Box the int
```

Unboxing reverses the operation, by casting the object back to the original value type:

```
int y = (int)obj; // Unbox the int
```

Unboxing requires an explicit cast. The runtime checks that the stated value type matches the actual object type, and throws an InvalidCastException if the check fails.

For instance, the following throws an exception, because long does not exactly match int:

```
object obj = 9;           // 9 is inferred to be of type int
long x = (long) obj;      // InvalidCastException
```

The following succeeds, however:

```
object obj = 9;
long x = (int) obj;
```

As does this:

```
object obj = 3.5;             // 3.5 is inferred to be of type double
int x = (int) (double) obj;   // x is now 3
```

The GetType Method and typeof Operator

All types in C# are represented at runtime with an instance of System.Type. There are two basic ways to get a System.Type object:

- Call GetType on the instance.
- Use the typeof operator on a type name.

GetType is evaluated at runtime; **typeof** is evaluated statically at compile time (when generic type parameters are involved, it's resolved by the Just-In-Time compiler).

System.Type has properties for such things as the type's name, assembly, base type, and so on. For example:

```
using System;

public class Point { public int X, Y; }

class Test
{
  static void Main()
  {
    Point p = new Point();
    Console.WriteLine (p.GetType().Name);             // Point
    Console.WriteLine (typeof (Point).Name);          // Point
    Console.WriteLine (p.GetType() == typeof(Point)); // True
    Console.WriteLine (p.X.GetType().Name);           // Int32
    Console.WriteLine (p.Y.GetType().FullName);       // System.Int32
  }
}
```

Structs

A struct is similar to a class, with the following key differences:

- A struct is a value type, whereas a class is a reference type.
- A struct does not support inheritance

.

A struct can have all the members a class can, except the following:

- A parameter less constructor
- Field initializers
- A finalizer
- Virtual or protected members

Here is an example of declaring and calling struct:

```
public struct Point
{
  int x, y;
  public Point (int x, int y) { this.x = x; this.y = y; }
}

...
Point p1 = new Point ();       // p1.x and p1.y will be 0
Point p2 = new Point (1, 1);   // p1.x and p1.y will be 1
```

The next example generates three compile-time errors:

```
public struct Point
{
  int x = 1;                          // Illegal: field initializer
  int y;
  public Point() {}                   // Illegal: parameterless constructor
  public Point (int x) {this.x = x;}  // Illegal: must assign field y
}
```

Changing `struct` to `class` makes this example legal.

Access Modifiers

Access modifiers in C# are used to specify the scope of accessibility of a member of a class or type of the class itself. For example, a public class is accessible to everyone without any restrictions, while an internal class may be accessible to the assembly only.

Access Modifiers	Inside Assembly		Outside Assembly	
	With Inheritance	With Type	With Inheritance	With Type
Public	✓	✓	✓	✓
Private	X	X	X	X
Protected	✓	X	✓	X
Internal	✓	✓	X	X
Protected Internal	✓	✓	✓	X

Access modifiers are an integral part of object-oriented programming. Access modifiers are used to implement encapsulation of OOP. Access modifiers allow you to define who does or who doesn't have access to certain features.

In C# there are 6 different types of Access Modifiers.

Modifier	Description
public	There are no restrictions on accessing public members.
private	Access is limited to within the class definition. This is the default access modifier type if none is formally specified
protected	Access is limited to within the class definition and any class that inherits from the class
internal	Access is limited exclusively to classes defined within the current project assembly
protected internal	Access is limited to the current assembly and types derived from the containing class. All members in current project and all members in derived class can access the variables.
private protected	Access is limited to the containing class or types derived from the containing class within the current assembly.

Examples

`Class2` is accessible from outside its assembly; `Class1` is not:

```
class Class1 {}                 // Class1 is internal (default)
public class Class2 {}
```

`ClassB` exposes field `x` to other types in the same assembly; `ClassA` does not:

```
class ClassA { int x;          } // x is private (default)
class ClassB { internal int x; }
```

Functions within `Subclass` can call `Bar` but not `Foo`:

```
class BaseClass
{
  void Foo()           {}        // Foo is private (default)
  protected void Bar() {}
}

class Subclass : BaseClass
{
  void Test1() { Foo(); }       // Error - cannot access Foo
  void Test2() { Bar(); }       // OK
}
```

Restrictions on Access Modifiers

```
class BaseClass             { protected virtual  void Foo() {} }
class Subclass1 : BaseClass { protected override void Foo() {} }  // OK
class Subclass2 : BaseClass { public    override void Foo() {} }  // Error
```

(An exception is when overriding a `protected internal` method in another assembly, in which case the override must simply be `protected`.)

The compiler prevents any inconsistent use of access modifiers. For example, a subclass itself can be less accessible than a base class, but not more:

```
internal class A {}
public class B : A {}          // Error
```

Enums in C#

Enum in C# language is a value type with a set of related named constants often referred to as an enumerator list. The enum keyword is used to declare an enumeration. It is a primitive data type, which is user-defined. Enums type can be an integer (float, int, byte, double etc.) but if you use beside int, it has to be cast.

Enum is used to create numeric constants in .NET framework. All member of the enum are of enum type. There must be a numeric value for each enum type.

The default underlying type of the enumeration elements is int. By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1.

```
// making an enumerator 'month'
enum month
{

    // following are the data members
    jan,
    feb,
    mar,
    apr,
    may

}

class Program {

    // Main Method
    static void Main(string[] args)
    {

        // getting the integer values of data members..
        Console.WriteLine("The value of jan in month " +
                          "enum is " + (int)month.jan);
        Console.WriteLine("The value of feb in month " +
                          "enum is " + (int)month.feb);
        Console.WriteLine("The value of mar in month " +
                          "enum is " + (int)month.mar);
        Console.WriteLine("The value of apr in month " +
                          "enum is " + (int)month.apr);
        Console.WriteLine("The value of may in month " +
                          "enum is " + (int)month.may);
    }
}
```

Output

```
The value of jan in month enum is 0
The value of feb in month enum is 1
The value of mar in month enum is 2
The value of apr in month enum is 3
The value of may in month enum is 4
```

Generics

Generics in C# and .NET procedure many of the benefits of strongly-typed collections as well as provide a higher quality of and a performance boost for code.
Generics are very similar to C++ templates but having a slight difference in such a way that the source code of C++ templates is required when a template is instantiated with a specific type and .NET Generics are not limited to classes only. In fact, they can also be implemented with Interfaces, Delegates and Methods.
The detailed specification for each collection is found under the **System.Collection.Generic namespace.**

Generic Classes

The Generic class can be defined by putting the <T> sign after the class name. It isn't mandatory to put the "T" word in the Generic type definition. You can use any word in the TestClass<> class declaration.

```
public class TestClass<T> { }
```

The **System.Collection.Generic** namespace also defines a number of classes that implement many of these key interfaces. The following table describes the core class types of this namespace.

Generic class	Description
Collection<T>	The basis for a generic collection Comparer compares two generic objects for equality
Dictionary<TKey, TValue>	A generic collection of name/value pairs
List<T>	A dynamically resizable list of Items
Queue<T>	A generic implementation of a first-in, first-out (FIFO) list
Stack<T>	A generic implementation of a last-in, first-out (LIFO) list

```
using System.Collections.Generic;
class Test<T>
    {
        T[] t=new T[5];
        int count = 0;
        public void addItem(T item)
        {
            if (count < 5)
            {
                t[count] = item;
                count++;
            }
            else
            {
                Console.WriteLine("Overflow exists");
            }
        }
```

```
    public void displayItem()
    {
        for (int i = 0; i < count; i++)
        {
        Console.WriteLine("Item at index {0} is {1}",i,t[i]);
        }
    }
}

class GenericEx
{
    static void Main()
    {
        Test<int> obj = new Test<int>();
        obj.addItem(10);
        obj.addItem(20);
        obj.addItem(30);
        obj.addItem(40);
        obj.addItem(50);
        //obj.addItem(60);  //overflow exists
        obj.displayItem();
        Console.ReadKey();
    }
}
```

Output

```
Item at index 0 is 10
Item at index 1 is 20
Item at index 2 is 30
Item at index 3 is 40
Item at index 4 is 50
```

Generic Methods

The objective of this example is to build a swap method that can operate on any possible data type (value-based or reference-based) using a single type parameter. Due to the nature of swapping algorithms, the incoming parameters will be sent by reference via ref keyword.

```
using System.Collections.Generic;
class Program
{
    //Generic method
    static void Swap<T>(ref T a, ref T b)
    {
        T temp;
        temp = a;
        a = b;
        b = temp;
    }
    static void Main(string[] args)
    {
        // Swap of two integers.
        int a = 40, b = 60;
        Console.WriteLine("Before swap: {0}, {1}", a, b);

        Swap<int>(ref a, ref b);

        Console.WriteLine("After swap: {0}, {1}", a, b);

        Console.ReadLine();
    }
}
```

Output

```
Before swap: 40, 60
After swap: 60, 40
```

Dictionary

Dictionaries are also known as maps or hash tables. It represents a data structure that allows you to access an element based on a key. One of the significant features of a dictionary is faster lookup; you can add or remove items without the performance overhead.

.Net offers several dictionary classes, for instance **Dictionary<TKey, TValue>.** The type parameters TKey and TValue represent the types of the keys and the values it can store, respectively.

```
using System.Collections.Generic;
public class Program
    {
        static void Main(string[] args)
        {
            //define Dictionary collection
            Dictionary<int, string> dObj = new Dictionary<int,
                  string>(5);
            //add elements to Dictionary
            dObj.Add(1, 1, "Tom");
            dObj.Add(2, "John");
            dObj.Add(3, "Maria");
```

```
            dObj.Add(4, "Max");
            dObj.Add(5, "Ram");

            //print data
            for (int i = 1; i <= dObj.Count; i++)
            {
                Console.WriteLine(dObj[i]);
            }
            Console.ReadKey();
        }
    }
```

Queues

Queues are a special type of container that ensures the items are being accessed in a FIFO (first in, first out) manner. Queue collections are most appropriate for implementing messaging components. We can define a Queue collection object using the following syntax:

```
Queue qObj = new Queue();
```

The Queue collection property, methods and other specification definitions are found under the Sysyem.Collection namespace. The following table defines the key members;

System.Collection.Queue Members	Definition
Enqueue()	Add an object to the end of the queue.
Dequeue()	Removes an object from the beginning of the queue.
Peek()	Return the object at the beginning of the queue without removing it.

```
using System.Collections.Generic;
class Program {
        static void Main(string[] args) {
            Queue < string > queue1 = new Queue < string > ();
            queue1.Enqueue("MCA");
            queue1.Enqueue("MBA");
            queue1.Enqueue("BCA");
            queue1.Enqueue("BBA");

            Console.WriteLine("The elements in the queue are:");
            foreach(string s in queue1) {
                Console.WriteLine(s);
            }

            queue1.Dequeue(); //Removes the first element that
              enter in the queue here the first element is MCA
            queue1.Dequeue(); //Removes MBA
```

```
            Console.WriteLine("After removal the elements in the
                                    queue are:");
             foreach(string s in queue1) {
                 Console.WriteLine(s);
             }
        }
    }
```

Stacks

A Stack collection is an abstraction of LIFO (last in, first out). We can define a Stack collection object using the following syntax:

```
Stack qObj = new Stack();
```

The following table illustrates the key members of a stack;

System.Collection.Stack Members	Definition
Contains()	Returns true if a specific element is found in the collection.
Clear()	Removes all the elements of the collection.
Peek()	Previews the most recent element on the stack.
Push()	It pushes elements onto the stack.
Pop()	Return and remove the top elements of the stack.

```
class Program {
        static void Main(string[] args) {
            Stack < string > stack1 = newStack < string > ();
            stack1.Push("************");
            stack1.Push("MCA");
            stack1.Push("MBA");
            stack1.Push("BCA");
            stack1.Push("BBA");
            stack1.Push("***********");
            stack1.Push("**Courses**");
            stack1.Push("***********");
            Console.WriteLine("The elements in the stack1 are
                                  as:");
            foreach(string s in stack1) {
                Console.WriteLine(s);
            }

           //For remove/or pop the element pop() method is used
            stack1.Pop();
            stack1.Pop();
            stack1.Pop();
            Console.WriteLine("After removal/or pop the element
                the stack is as:");
            //the element that inserted in last is remove firstly.
```

```
            foreach(string s in stack1) {
                Console.WriteLine(s);
             }
        }
    }
```

List

List<T> class in C# represents a strongly typed list of objects. List<T> provides functionality to create a list of objects, find list items, sort list, search list, and manipulate list items. In List<T>, T is the type of objects.

Adding Elements

```
// Dynamic ArrayList with no size limit
            List<int> numberList = new List<int>();
            numberList.Add(32);
            numberList.Add(21);
            numberList.Add(45);
            numberList.Add(11);
            numberList.Add(89);
            // List of string
            List<string> authors = new List<string>(5);
            authors.Add("Mahesh Chand");
            authors.Add("Chris Love");
            authors.Add("Allen O'neill");
            authors.Add("Naveen Sharma");
            authors.Add("Monica Rathbun");
            authors.Add("David McCarter");

// Collection of string
            string[] animals = { "Cow", "Camel", "Elephant" };
            // Create a List and add a collection
            List<string> animalsList = new List<string>();
            animalsList.AddRange(animals);
            foreach (string a in animalsList)
                Console.WriteLine(a);
```

Remove Elements

```
// Remove an item
    authors.Remove("New Author1");

// Remove 3rd item
    authors.RemoveAt(3);

 // Remove all items
    authors.Clear();
```

Sorting

```
authors.Sort();
```

Other Methods

```
authors.Insert(1,"Shaijal"); //insert item at index 1
authors.Count;      //returns total items
```

Array List

C# ArrayList is a non-generic collection. The ArrayList class represents an array list and it can contain elements of any data types. The ArrayList class is defined in the System.Collections namespace. An ArrayList is dynamic array and grows automatically when new items are added to the collection.

```
ArrayList personList = new ArrayList();
```

Insertion

```
personList.Add("Sandeep");
```

Removal

```
// Remove an item
    personList.Remove("New Author1");

// Remove 3rd item
    personList.RemoveAt(3);

 // Remove all items
    personList.Clear();
```

Sorting

```
personList.Sort();
```

Other Methods

```
personList.Insert(1,"Shaijal"); //insert item at index 1
personList.Count;      //returns total items
```

Unit - 4 Advanced C# [14 Hrs]

Delegates; Events; Lambda Expressions; Exception Handling; Introduction to LINQ; Working with Databases; Web Applications using ASP.NET

Delegates

A delegate is an object that knows how to call a method.

A delegate type defines the kind of method that delegate instances can call. Specifically, it defines the method's return type and its parameter types.

Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class. It provides a way which tells which method is to be called when an event is triggered.

For example, if you click an ***Button*** on a form **(Windows Form application),** the program would call a specific method.

In simple words, it is a type that represents references to methods with a particular parameter list and return type and then calls the method in a program for execution when it is needed.

Declaring Delegates

Delegate type can be declared using the **delegate** keyword. Once a delegate is declared, delegate instance will refer and call those methods whose return type and parameter-list matches with the delegate declaration.

Syntax:

```
[modifier] delegate [return_type] [delegate_name] ([parameter_list]);
```

modifier*: It is the required modifier which defines the access of delegate and it is optional to use.*

delegate*: It is the keyword which is used to define the delegate.*

*return_**type**: It is the type of value returned by the methods which the delegate will be going to call. It can be void. A method must have the same return type as the delegate.*

delegate_name*: It is the user-defined name or identifier for the delegate.*

parameter_list*: This contains the parameters which are required by the method when called through the delegate.*

```
public delegate int DelegateTest(int x, int y, int z);
```

Note: A delegate will call only a method which agrees with its signature and return type. A method can be a static method associated with a class or can be an instance method associated with an object, it doesn't matter.

Instantiation & Invocation of Delegates

Once a delegate type is declared, a delegate object must be created with the **new** keyword and be associated with a particular method. When creating a delegate, the argument passed to the **new** expression is written similar to a method call, but without the arguments to the method. **For example**

```
[delegate_name]  [instance_name] = new
[delegate_name](calling_method_name);
```

```
DelegateTest obj = new DelegateTest (MyMethod);
       // here,
       // " DelegateTest" is delegate name.
       // " obj" is instance_name
       // " MyMethod" is the calling method.
```

The following defines a delegate type called **Transformer**:

```
delegate int Transformer (int x);
```

Transformer is **compatible** with any method with an **int return type and a single int parameter**, such as this:

```
static int Square (int x) { return x * x; }
```

or

```
static int Square (int x) => x * x;
```

Assigning a method to a delegate variable creates a delegate instance:

```
Transformer t = new Transformer (Square);
```

Or

```
Transformer t = Square;
```

which can be invoked in the same way as a method:

```
int answer = t(3); // answer is 9
```

Example of simple delegate is shown below:

```
public delegate int MyDelegate(int x);
    class DelegateTest
    {
        static int MyMethod(int x)
        {
            return x * x;
        }

        static void Main(string[] agrs)
        {
            MyDelegate del = new MyDelegate(MyMethod);
            int res = del(5); //25
            Console.WriteLine("Result is : "+res);
            Console.ReadKey();
        }
    }
```

Output:

```
Result is : 25
```

Example 2

```
using System;

delegate int NumberChanger(int n);

   class TestDelegate {
      static int num = 10;

      public static int AddNum(int p) {
         num += p;
         return num;
      }
      public static int MultNum(int q) {
         num *= q;
         return num;
      }
      public static int getNum() {
         return num;
      }
      static void Main(string[] args) {
         //create delegate instances
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultNum);

         //calling the methods using the delegate objects
         nc1(25);
         Console.WriteLine("Value of Num: {0}", getNum());
         nc2(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
```

Output:

```
Value of Num: 35
Value of Num: 175
```

Example 3

```
class Test
    {
        public delegate void addnum(int a, int b);
        public delegate void subnum(int a, int b);

        // method "sum"
        public void sum(int a, int b)
        {
            Console.WriteLine("(100 + 40) = {0}", a + b);
        }
```

```
    // method "subtract"
      public void subtract(int a, int b)
      {
          Console.WriteLine("(100 - 60) = {0}", a - b);
      }

      // Main Method
      public static void Main(String[] args)
      {
          addnum del_obj1 = new addnum(obj.sum);
          subnum del_obj2 = new subnum(obj.subtract);

          // pass the values to the methods by delegate object
          del_obj1(100, 40);
          del_obj2(100, 60);

          // These can be written as using
          // "Invoke" method
          // del_obj1.Invoke(100, 40);
          // del_obj2.Invoke(100, 60);
      }
}
```

Output:

```
(100 + 40) = 140
(100 - 60) = 40
```

Multicast Delegates

All delegate instances have multicast capability**. This means that a delegate instance can reference not just a single target method, but also a list of target methods**.
The + and += operators combine delegate instances. For example:

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

The last line is functionally the same as:

```
d = d + SomeMethod2;
```

Invoking d will now call both SomeMethod1 and SomeMethod2. **Delegates are invoked in the order they are added.**

The - and -= operators remove the right delegate operand from the left delegate operand. For example:

```
d -= SomeMethod1;
```

Invoking d will now cause only SomeMethod2 to be invoked.

Calling + or += on a delegate variable with a null value works, and it is equivalent to assigning the variable to a new value:

```
SomeDelegate d = null;
d += SomeMethod1; // Equivalent (when d is null) to d = SomeMethod1;
```

Similarly, calling -= on a delegate variable with a single target is equivalent to assigning null to that variable.

Example of multicasting delegates

```
using System;

delegate int NumberChanger(int n);
   class TestDelegate {
      static int num = 10;

      public static int AddNum(int p) {
         num += p;
         return num;
      }
      public static int MultNum(int q) {
         num *= q;
         return num;
      }
      public static int getNum() {
         return num;
      }
      static void Main(string[] args) {
         //create delegate instances
         NumberChanger nc;
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultNum);

         nc = nc1;
         nc += nc2;

         //calling multicast
         nc(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
```

Output

```
Value of Num: 75
```

Delegates Mapping with Instance and Static Method

Created ClassA contains instance & static method,

```
class A
{
    public void InstanceMethod()
    {
        Console.WriteLine("InstanceMethod");
    }

    public static void StaticMethod()
    {
        Console.WriteLine("StaticMethod");
    }
}
```

Above class methods are used using delegate,

```
class Program
{
    //declaration of delegate
    delegate void Del();


    static void Main(string[] args)
    {
        A objA = new A();

        //Instance Method associated with delegate instance
        Del d = objA.InstanceMethod;
        d();



        //Static method associated with same delegate instance
        d = A.StaticMethod;
        d();

        Console.ReadLine();

    }
```

Output

Instance Method
Static Method

So using delegate, we can associate instance and static method under same delegate instance.

Delegates Vs Interfaces in C#

S.N.	DELEGATE	INTERFACE
1	It could be a method only.	It contains both methods and properties.
2	It can be applied to one method at a time.	If a class implements an interface, then it will implement all the methods related to that interface.
3	If a delegate available in your scope you can use it.	Interface is used when your class implements that interface, otherwise not.
4	Delegates can be implemented any number of times.	Interface can be implemented only one time.
5	It is used to handling events.	It is not used for handling events.
6	When you access the method using delegates you do not require any access to the object of the class where the method is defined.	When you access the method you need the object of the class which implemented an interface.
7	It does not support inheritance.	It supports inheritance.
8	It created at run time.	It created at compile time.
9	It can implement any method that provides the same signature with the given delegate.	If the method of interface implemented, then the same name and signature method override.
10	It can wrap any method whose signature is similar to the delegate and does not consider which from class it belongs.	A class can implement any number of interfaces, but can only override those methods which belongs to the interfaces.

Delegate Compatibility

Type compatibility

Delegate types are all incompatible with one another, even if their signatures are the same:

```
delegate void D1();
delegate void D2();
...

D1 d1 = Method1;
D2 d2 = d1;                          // Compile-time error
```

The following, however, is permitted:

```
D2 d2 = new D2 (d1);
```

Delegate instances are considered equal if they have the same method targets:

```
delegate void D();
...

D d1 = Method1;
D d2 = Method1;
Console.WriteLine (d1 == d2);         // True
```

Multicast delegates are considered equal if they reference the same methods *in the same order.*

Parameter compatibility

When you call a method, you can supply arguments that have more specific types than the parameters of that method. This is ordinary polymorphic behaviour. For exactly the same reason, a delegate can have more specific parameter types than its method target. This is called **contravariance**.

Here's an example:

```
delegate void StringAction (string s);

class Test
{
  static void Main()
  {
    StringAction sa = new StringAction (ActOnObject);
    sa ("hello");
  }

  static void ActOnObject (object o) => Console.WriteLine (o);   // hello
}
```

In this case, the String Action is invoked with an argument of type string. When the argument is then relayed to the target method, the argument gets implicitly up cast to an object.

Return type compatibility

If you call a method, you may get back a type that is more specific than what you asked for. This is ordinary polymorphic behaviour. For exactly the same reason, a delegate's target method may return a more specific type than described by the dele- gate. This is called **covariance**. For example:

```
delegate object ObjectRetriever();

class Test
{
  static void Main()
  {
    ObjectRetriever o = new ObjectRetriever (RetrieveString);
    object result = o();
    Console.WriteLine (result);      // hello
  }
  static string RetrieveString() => "hello";
}
```

Generic Delegate Types

A delegate type may contain generic type parameters. For example:

```
public delegate T Transformer (T arg);
```

With this definition, we can write a generalized Transform utility method that works on any type:

```
public class Util
{
  public static void Transform<T> (T[] values, Transformer<T> t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t (values[i]);
  }
}

class Test
{
  static void Main()
  {
    int[] values = { 1, 2, 3 };
    Util.Transform (values, Square);      // Hook in Square
    foreach (int i in values)
      Console.Write (i + "  ");           // 1   4   9
  }

  static int Square (int x) => x * x;
}
```

Func and Action Delegates

The Func and Action generic delegates were introduced in the .NET Framework version 3.5. Whenever we want to use delegates in our examples or applications, typically we use the following procedure:

- Define a custom delegate that matches the format of the method.
- Create an instance of a delegate and point it to a method.
- Invoke the method.

But, using these 2 Generics delegates we can simply eliminate the above procedure.

Since both the delegates are generic, you will need to specify the underlying types of each parameter as well while pointing it to a function. For for example Action<type,type,type……>

Action<>

- This Action<> generic delegate; points to a method that takes up to 16 Parameters and returns void.

Func<>

- The generic Func<> delegate is used when we want to point to a method that returns a value.
- This delegate can point to a method that takes up to 16 Parameters and returns a value.

- Always remember that the final parameter of Func<> is always the return value of the method. (For example, Func< int, int, string>, this version of the Func<> delegate will take 2 int parameters and returns a string value.)

Example is shown below

```
class MethodCollections
    {
        //Methods that takes parameters but returns nothing:

        public static void PrintText()
        {
            Console.WriteLine("Text Printed with the help of Action");
        }
        public static void PrintNumbers(int start, int target)
        {
            for (int i = start; i <= target; i++)
            {
                Console.Write(" {0}",i);
            }
            Console.WriteLine();
        }
        public static void Print(string message)
        {
            Console.WriteLine(message);
        }

        //Methods that takes parameters and returns a value:

        public static int Addition(int a, int b)
        {
            return a + b;
        }

        public static string DisplayAddition(int a, int b)
        {
           return string.Format("Addition of {0} and {1} is {2}",a,b,a+b);

        }

        public static string SHowCompleteName(string firstName, string las
                  tName)
        {
          return string.Format("Your Name is {0} {1}",firstName,lastName);

        }
        public static int ShowNumber()
        {
            Random r = new Random();
            return r.Next();
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            Action printText = new Action(MethodCollections.PrintText);
            Action<string> print = new Action<string>(MethodCollections.Pr
                                                int);
            Action<int, int> printNumber = new Action<int, int>(MethodColl
                                                    ections.PrintNumbers);

            Func<int, int,int> add1 = new Func<int, int, int>(MethodCollec
                                                    tions.Addition);
            Func<int, int, string> add2 = new Func<int, int, string>(Metho
                                        dCollections.DisplayAddition);
            Func<string, string, string> completeName = new Func<string, s
                  tring, string>(MethodCollections.SHowCompleteName);
           Func<int> random = new Func<int>(MethodCollections.ShowNumber);

           Console.WriteLine("\n***************** Action<> Delegate Method
                                        ***************\n");
            printText();    //Parameter: 0 , Returns: nothing
            print("Abhishek");  //Parameter: 1 , Returns: nothing
            printNumber(5, 20); //Parameter: 2 , Returns: nothing
            Console.WriteLine();
            Console.WriteLine("***************** Func<> Delegate Methods **
                                        ***************\n");
            int addition = add1(2, 5);  //Parameter: 2 , Returns: int
            string addition2 = add2(5, 8);  //Parameter:2 ,Returns: string

            string name = completeName("Abhishek", "Yadav"); //Parameter:2
                                , Returns: string
           int randomNumbers = random();   ////Parameter: 0 , Returns: int
            Console.WriteLine("Addition: {0}",addition);
            Console.WriteLine(addition2);
            Console.WriteLine(name);
            Console.WriteLine("Random Number is: {0}",randomNumbers);

            Console.ReadLine();
        }
    }
```

```
****************** Action<> Delegate Methods ****************

Text Printed with the help of Action
Abhishek
 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

****************** Func<> Delegate Methods ******************

Addition: 7
Addition of 5 and 8 is 13
Your Name is Abhishek Yadav
Random Number is: 1848661255
```

Events

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

The class that sends or raises an event is called a Publisher and class that receives or handle the event is called "Subscriber".

Following are the key points about Event,

1. Event Handlers in C# return void and take two parameters.
2. The First parameter of Event - Source of Event means publishing object.
3. The Second parameter of Event - Object derived from EventArgs.
4. The publishers determines when an event is raised and the subscriber determines what action is taken in response.
5. An Event can have so many subscribers.
6. Events are basically used for the single user action like button click.
7. If an Event has multiple subscribers then event handlers are invoked synchronously.

Declaring Events

To declare an event inside a class, first of all, you must declare a delegate type for the even as:

```
public delegate string MyDelegate(string str);
```

then, declare the event using the **event** keyword –

```
event MyDelegate delg;
```

The preceding code defines a delegate named MyDelegate and an event named delg, which invokes the delegate when it is raised.

To declare an event inside a class, first a Delegate type for the Event must be declared like below:

```
public delegate void MyEventHandler(object sender, EventArgs e);
```

Defining an event is a two-step process.

- First, you need to define a delegate type that will hold the list of methods to be called when the event is fired.
- Next, you declare an event using the event keyword.

To illustrate the event, we are creating a console application. In this iteration, we will define an event to add that is associated to a single delegate **DelEventHandler**.

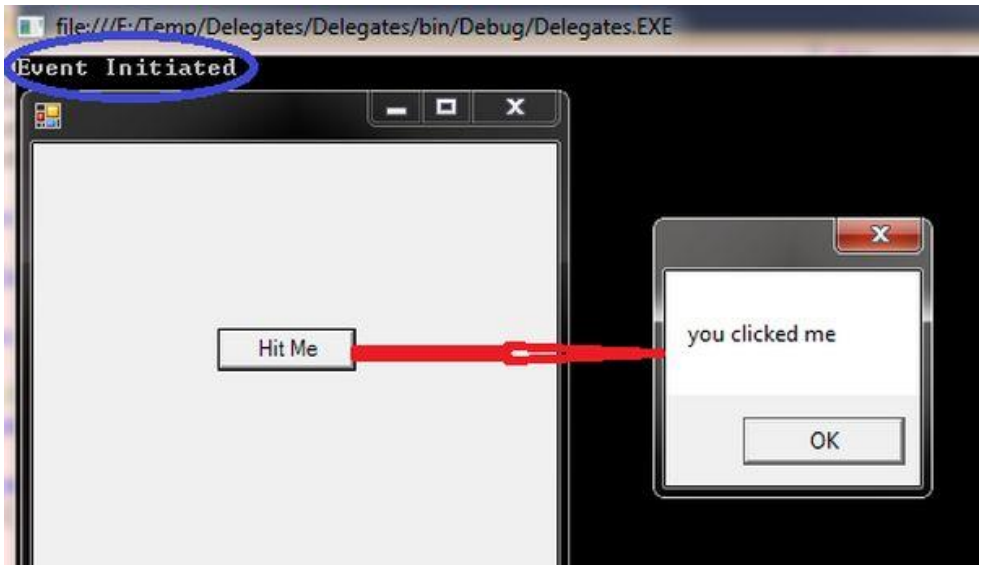
```
using System;
   public delegate void DelEventHandler();

    class Program
    {
        public static event DelEventHandler add;
```

```
        static void USA()
          {
              Console.WriteLine("USA");
          }

          static void India()
          {
              Console.WriteLine("India");
          }

          static void England()
          {
              Console.WriteLine("England");
          }


          static void Main(string[] args)
          {

              add += new DelEventHandler(USA);
              add += new DelEventHandler(India);
              add += new DelEventHandler(England);
              add.Invoke();

              Console.ReadLine();
          }

      }
```

Implementing event in a button click

```
using System;
using System.Drawing;
using System.Windows.Forms;

    //custom delegate
    public delegate void DelEventHandler();

    class Program :Form
    {
        //custom event
        public event DelEventHandler add;

        public Program()
        {
            // design a button over form
            Button btn = new Button();
            btn.Parent = this;
            btn.Text = "Hit Me";
            btn.Location = new Point(100,100);

            //Event handler is assigned to
```

```
            // the button click event
            btn.Click += new EventHandler(onClcik);
            add += new DelEventHandler(Initiate);

            //invoke the event
            add();
        }
        //call when event is fired
        public void Initiate()
        {
            Console.WriteLine("Event Initiated");
        }

        //call when button clicked
        public void onClcik(object sender, EventArgs e)
        {
            MessageBox.Show("You clicked me");
        }
        static void Main(string[] args)
        {
            Application.Run(new Program());

            Console.ReadLine();
        }
}
```



Can we use Events without Delegate?

No, Events use Delegates internally. Events are encapsulation over Delegates. There is already defined Delegate "EventHandler" that can be used like below:

public event EventHandler MyEvents;

So, it also used Delegate Internally.

Anonymous Method in C#

An anonymous method is a method which doesn't contain any name which is introduced in C# 2.0. It is useful when the user wants to create an inline method and also wants to pass parameter in the anonymous method like other methods.
An Anonymous method is defined using the delegate keyword and the user can assign this method to a variable of the delegate type.

```
delegate(parameter_list){
    // Code..
};
```

Example:

```
using System;

class GFG {

    public delegate void petanim(string pet);

    // Main method
    static public void Main()
    {

        // An anonymous method with one parameter
        petanim p = delegate(string mypet)
        {
            Console.WriteLine("My favorite pet is: {0}",
                                                mypet);
        };
        p("Dog");
    }
}
```

Output:

My favorite pet is: Dog

You can also use an anonymous method as an event handler.

```
MyButton.Click += delegate(Object obj, EventArgs ev)
{
    System.Windows.Forms.MessageBox.Show("Complete without
                 error...!!");
}
```

Lambda Expressions

Lambda expressions in C# are used like anonymous functions, with the difference that in Lambda expressions you don't need to specify the type of the value that you input thus making it more flexible to use.

The '=>' is the lambda operator which is used in all lambda expressions. The Lambda expression is divided into two parts, the left side is the input and the right is the expression.

The Lambda Expressions can be of two types:

1. **Expression Lambda:** Consists of the input and the expression.
   Syntax:
   input => expression;
2. **Statement Lambda:** Consists of the input and a set of statements to be executed. It can be used along with delegates.
   Syntax:
   input => { statements };

Basic example of lambda expression:

```
class LambdaTest
    {
        static int test1() => 5;
        static int test2(int x) => x + 10;

        static void Main(string[] args)
        {
            int x=test1();
            int res = test2(x);
            Console.WriteLine("Result is: "+res);
        }
    }
```

Output:

```
Result is: 15
```

Unlike an expression lambda, a statement lambda can contain multiple statements separated by semicolons. It is used with delegates.

```
delegate void ModifyInt(int input);

ModifyInt addOneAndTellMe = x =>
{
    int result = x + 1;
    Console.WriteLine(result);
};
```

Exception Handling

A try statement specifies a code block subject to error-handling or clean-up code. The try block must be followed by a catch block, a finally block, or both. The catch block executes when an error occurs in the try block. The finally block executes after execution leaves the try block (or if present, the catch block), to per- form clean-up code, whether or not an error occurred.

A catch block has access to an Exception object that contains information about the error. You use a catch block to either compensate for the error or re throw the exception. You re throw an exception if you merely want to log the problem, or if you want to re throw a new, higher-level exception type.

A finally block adds determinism to your program: the CLR endeavours to always execute it. It's useful for clean-up tasks such as closing network connections.

A try statement looks like this:

```
try
{
  ... // exception may get thrown within execution of this block
}
catch (ExceptionA ex)
{
  ... // handle exception of type ExceptionA
}
catch (ExceptionB ex)
{
  ... // handle exception of type ExceptionB
}
finally
{
  ... // cleanup code
}
```

Consider the following program:

```
class Test
{
  static int Calc (int x) => 10 / x;

  static void Main()
  {
    int y = Calc (0);
    Console.WriteLine (y);
  }
}
```

Because x is zero, the runtime throws a DivideByZeroException, and our program terminates. We can prevent this by catching the exception as follows:

```
class Test
{
  static int Calc (int x) => 10 / x;

  static void Main()
  {
    try
    {
      int y = Calc (0);

      Console.WriteLine (y);
    }
    catch (DivideByZeroException ex)
    {
      Console.WriteLine ("x cannot be zero");
    }
    Console.WriteLine ("program completed");
  }
}

OUTPUT:
x cannot be zero
program completed
```

The catch Clause

A catch clause specifies what type of exception to catch. This must either be System.Exception or a subclass of System.Exception.
You can handle multiple exception types with multiple catch clauses:

```
class Test
{
  static void Main (string[] args)
  {
    try
    {
      byte b = byte.Parse (args[0]);
      Console.WriteLine (b);
    }
    catch (IndexOutOfRangeException ex)
    {
      Console.WriteLine ("Please provide at least one argument");
    }
    catch (FormatException ex)
    {
      Console.WriteLine ("That's not a number!");
    }
    catch (OverflowException ex)
    {
      Console.WriteLine ("You've given me more than a byte!");
    }
  }
}
```

The finally Block

A finally block always executes—whether or not an exception is thrown and whether or not the try block runs to completion. finally blocks are typically used for clean-up code.
A finally block executes either:

- After a catch block finishes
- After control leaves the try block because of a jump statement (e.g., return or goto)
- After the try block ends

Throwing Exceptions

Exceptions can be thrown either by the runtime or in user code. In this example, Display throws a System.ArgumentNullException:

```
class Test
{
  static void Display (string name)
  {
    if (name == null)
      throw new ArgumentNullException (nameof (name));

    Console.WriteLine (name);
  }

  static void Main()
  {

     try { Display (null); }
     catch (ArgumentNullException ex)
     {
       Console.WriteLine ("Caught the exception");
     }
   }
}
```

Re-throwing an exception

You can capture and re-throw an exception as follows:

```
try {  ...  }
catch (Exception ex)
{
  // Log error
  ...
  throw;          // Rethrow same exception
}
```

Common Exception Types

System.ArgumentException
Thrown when a function is called with a bogus argument. This generally indi- cates a program bug.

System.ArgumentNullException
Subclass of ArgumentException that's thrown when a function argument is (unexpectedly) null.

System.ArgumentOutOfRangeException
Subclass of ArgumentException that's thrown when a (usually numeric) argu- ment is too big or too small. For example, this is thrown when passing a nega- tive number into a function that accepts only positive values.

System.InvalidOperationException
Thrown when the state of an object is unsuitable for a method to successfully execute, regardless of any particular argument values. Examples include read- ing an unopened file or getting the next element from an enumerator where the underlying list has been modified partway through the iteration.

System.NotSupportedException
Thrown to indicate that a particular functionality is not supported. A good example is calling the Add method on a collection for which IsReadOnly returns true.

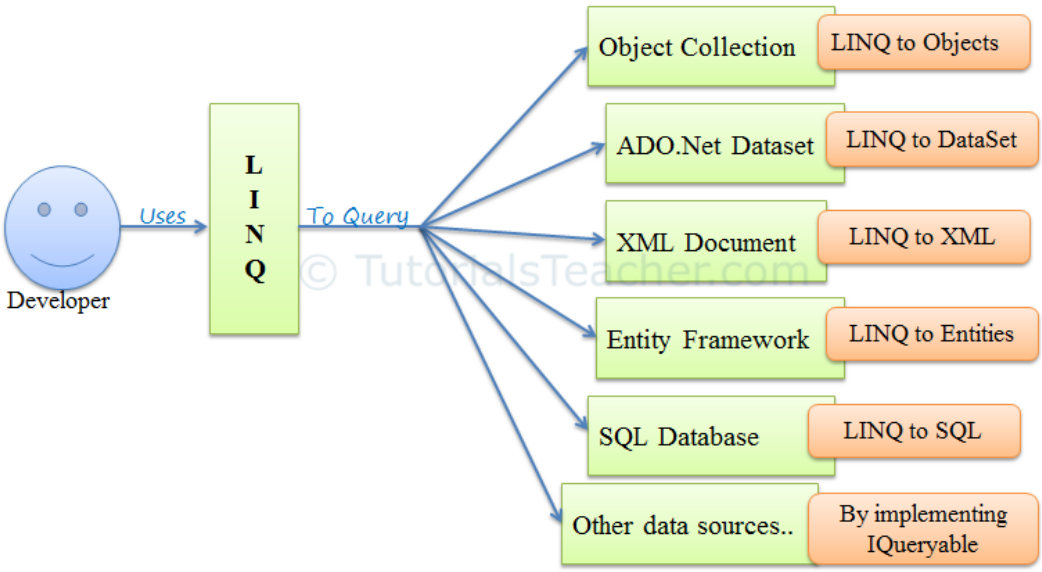
System.NotImplementedException
Thrown to indicate that a function has not yet been implemented.

System.ObjectDisposedException
Thrown when the object upon which the function is called has been disposed.

Introduction to LINQ

LINQ (Language Integrated Query) is uniform query syntax in C# to retrieve data from different sources and formats. It is integrated in C#, thereby eliminating the mismatch between programming languages and databases, as well as providing a single querying interface for different types of data sources.

For example, SQL is a Structured Query Language used to save and retrieve data from a database. In the same way, LINQ is a structured query syntax built in C# to retrieve data from different types of data sources such as collections, ADO.Net DataSet, XML Docs, web service and MS SQL Server and other databases.



LINQ queries return results as objects. It enables you to uses object-oriented approach on the result set and not to worry about transforming different formats of results into objects.

The following example demonstrates a simple LINQ query that gets all strings from an array which contains 'a'.

Example: LINQ Query to Array

```
// Data source
string[] names = {"Bill", "Steve", "James", "Mohan" };
// LINQ Query
var myLinqQuery = from name in names
                  where name.Contains('a')
                  select name;
// Query execution
foreach(string name in myLinqQuery)
    Console.Write(name + " ");
```

Example: LINQ Query to List

```
// string collection
List<string> stringList = new List<string>() {
    "C# Tutorials",
    "VB.NET Tutorials",
    "Learn C++",
    "MVC Tutorials" ,
    "Java"
};

var result = from s in stringList
             where s.Contains("Tutorials")
             select s;

foreach(string value in result)
    Console.Write(value + " ");
```

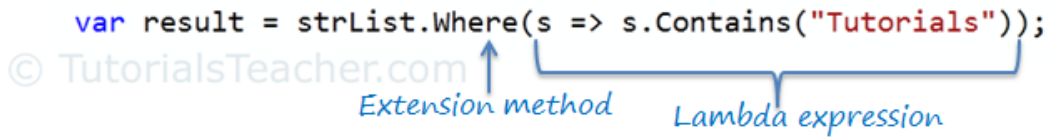
LINQ Method

The following is a sample LINQ method syntax query that returns a collection of strings which contains a word "Tutorials". **We use lambda expression for this purpose**.

Example: LINQ Method Syntax in C#

```
// string collection
List<string> stringList = new List<string>() {
    "C# Tutorials",
    "VB.NET Tutorials",
    "Learn C++",
    "MVC Tutorials" ,
    "Java"
};

// LINQ Query Syntax
var result = stringList.Where(s => s.Contains("Tutorials"));
```

The following figure illustrates the structure of LINQ method syntax.



Use of lambda Expression to LINQ

Example 1

```
using System;
using System.Collections.Generic;
using System.Linq;
public class demo
{
    public static void Main()
    {
        List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };
        List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);
```

```
        foreach (var num in evenNumbers)
        {
            Console.Write("{0} ", num);
        }
        Console.WriteLine();
        Console.Read();

    }
}
```

Output:

2 4 6

Example 2

```
using System;
using System.Collections.Generic;
using System.Linq;
class Dog
{
    public string Name { get; set; }
    public int Age { get; set; }
}
class demo{
    static void Main()
    {
        List<Dog> dogs = new List<Dog>() {
            new Dog { Name = "Rex", Age = 4 },
            new Dog { Name = "Sean", Age = 0 },
            new Dog { Name = "Stacy", Age = 3 }
         };
         var names = dogs.Select(x => x.Name);
         foreach (var name in names)
         {
             Console.WriteLine(name);

         }
         Console.Read();
    }
}
```

Output:

Rex
Sean
Stacy

Sorting using a lambda expression

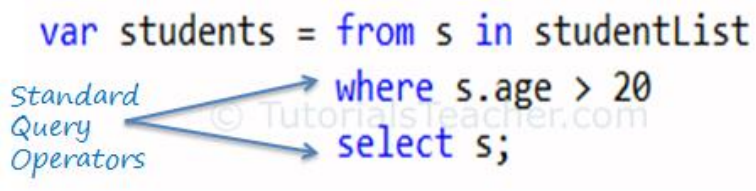
```
var sortedDogs = dogs.OrderByDescending(x => x.Age);
foreach (var dog in sortedDogs)
{
  Console.WriteLine("Dog {0} is {1} years old.", dog.Name, dog.Age
);
}
```

Output:

```
Dog Rex is 4 years old.
Dog Stacy is 3 years old.
Dog Sean is 0 years old.
```

LINQ Operators

Classification	LINQ Operators
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy, ToLookup
Join	GroupJoin, Join
Projection	Select, SelectMany
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum
Quantifiers	All, Any, Contains
Elements	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Set	Distinct, Except, Intersect, Union
Partitioning	Skip, SkipWhile, Take, TakeWhile
Concatenation	Concat
Equality	Equals,SequenceEqual
Generation	DefaultEmpty, Empty, Range, Repeat
Conversion	AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList



Some examples of LINQ Query

Where Condition – Example 1

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace BCA
{
    class LinqTest
    {
        static void Main(string[] args)
        {
            List<string> names = new List<string>() {
                  "Ram","Shyam","Hari","Gita"};
            //single condition
            //var result = names.Where(s => s.Contains("Ram"));

            //Multiple Condition
            var result = names.Where(s=>s.Contains("Ram") ||
                              s.Contains("Gita"));
            foreach (string val in result)
            {
                Console.WriteLine(val);
            }

            Console.ReadLine();
        }
    }
}
```

Output:

Ram
Gita

Where Condition – Example 2 (with object list)

```
class Student
    {
        public int sid { get;set;}
        public string name { get; set; }
        public string address { get; set; }

        public Student(int sid, string name, string address)
        {
            this.sid = sid;
            this.name = name;
            this.address = address;
        }
    }

    class LinqTest
    {
        static void Main(string[] args)
```

```
        {
            List<Student> mylist = new List<Student>(){
            new Student(1,"Ram","Btm"),
            new Student(2, "Hari", "Ktm"),
            new Student(3,"Shyam","Btm"),
            new Student(4, "Gita", "Ktm")
            };

            //var result = mylist.Where(s=>s.address.Contains("Btm"));
            var result = mylist.Where(s => s.address.Equals("Btm") &&
                              s.sid.Equals(1));
            Console.WriteLine("Sid\tName\tAddress");
            foreach (var res in result)
            {

Console.WriteLine(res.sid+"\t"+res.name+"\t"+res.address);
            }
            Console.ReadLine();
        }
    }
```

Output:

Sid	Name	Address
1	Ram	Btm

Joining multiple lists – (join, concat, union)

```
class LinqTest
    {
        static void Main(string[] args)
        {
            List<string> names = new List<string>() {
                      "Ram","Shyam","Hari"};
            List<string> address = new List<string>()
                      {"Btm","Ktm","Btm" };
            /*using join
            var result = names.Join(address,
                      str1 => str1,
                      str2 => str2,
                      (str1, str2) => str1);*/
            /*using concat
            var result = names.Concat(address);*/

            //using union
            var result = names.Union(address);
            foreach (var res in result)
            {
                Console.WriteLine(res);
            }

            Console.ReadLine();
        }    }
```

Using aggregate functions – Example 1

```
class LinqTest
    {
        static void Main(string[] args)
        {
            List<int> marks = new List<int>() { 10,30,50,20,5};
            int max = marks.Max();
            int min = marks.Min();
            int sum = marks.Sum();
            int total = marks.Count();

            Console.WriteLine("Maximum marks="+max);
            Console.WriteLine("Minimum marks=" + min);
            Console.WriteLine("Sum of marks=" + sum);
            Console.WriteLine("Total Count=" + total);

            Console.ReadLine();
        }
    }
```

Using aggregate functions – Example 2 (Object List)

```
class Student
    {
        public int sid { get; set; }
        public string name { get; set; }
        public string address { get; set; }

        public Student(int sid, string name, string address)
        {
            this.sid = sid;
            this.name = name;
            this.address = address;
        }
    }

    class LinqTest
    {
        static void Main(string[] args)
        {
            List<Student> mylist = new List<Student>(){
            new Student(1,"Ram","Btm"),
            new Student(2, "Hari", "Ktm"),
            new Student(3,"Shyam","Btm"),
            new Student(4, "Gita", "Ktm")
            };

            int maxId = mylist.Max(s=>s.sid);
            int count = mylist.Count();
            Console.WriteLine("Max Id="+maxId);
```

```
            Console.WriteLine("Total Students="+count);

            Console.ReadLine();
        }
    }
```

Output

```
Max Id=4
Total Students=4
```

Using Order By

```
class Student
    {
        public int sid { get; set; }
        public string name { get; set; }
        public string address { get; set; }

        public Student(int sid, string name, string address)
        {
            this.sid = sid;
            this.name = name;
            this.address = address;
        }
    }

    class LinqTest
    {
        static void Main(string[] args)
        {
            List<Student> mylist = new List<Student>(){
            new Student(1,"Ram","Btm"),
            new Student(2, "Hari", "Ktm"),
            new Student(3,"Shyam","Btm"),
            new Student(4, "Gita", "Ktm")
            };

            //var result = mylist.OrderBy(s=>s.name);
            //var result = mylist.OrderByDescending(s => s.name);

            //select name and address of student order by name in
                ascending where address is Ktm
            var result = mylist.Where(s=>s.address.Equals("Ktm")).
                          OrderBy(s=>s.name);

            Console.WriteLine("Name\tAddress");
            foreach (var res in result)
            {
                Console.WriteLine(res.name + "\t" + res.address);
            }
```

```
            Console.ReadLine();
        }
    }
```

Output:

Name	Address
Gita	Ktm
Hari	Ktm

Using Group By

```
class Student
    {
        public int sid { get; set; }
        public string name { get; set; }
        public string address { get; set; }

        public Student(int sid, string name, string address)
        {
            this.sid = sid;
            this.name = name;
            this.address = address;
        }
    }

    class LinqTest
    {
        static void Main(string[] args)
        {
            List<Student> mylist = new List<Student>(){
            new Student(1,"Ram","Btm"),
            new Student(2, "Hari", "Ktm"),
            new Student(3,"Shyam","Btm"),
            new Student(4, "Gita", "Ktm")
            };

            //select records group by address
            var groupResult = mylist.GroupBy(s=>s.address);
            foreach (var result in groupResult)
            {
                Console.WriteLine("Group Key: " + result.Key);
                      //Each group has key
                Console.WriteLine("Sid\tName\tAddress");
                foreach (var res in result)
                {

                      Console.WriteLine(res.sid+"\t"+res.name+"\t"
                      + res.address);
                }
```

```
            }
            Console.ReadLine();
        }
    }
```

Output:

Group Key: Btm

Sid	Name	Address
1	Ram	Btm
3	Shyam	Btm

Group Key: Ktm

Sid	Name	Address
2	Hari	Ktm
4	Gita	Ktm

Working with Databases

Comparison between ADO and ADO.NET:

ADO is a Microsoft technology. It stands for ActiveX Data Objects. It is a Microsoft Active-X component. ADO is automatically installed with Microsoft IIS. It is a programming interface to access data in a database

ADO.NET is a set of classes that expose data access services for .NET Framework programmers. ADO.NET provides a rich set of components for creating distributed, data-sharing applications. It is an integral part of the .NET Framework, providing access to relational, XML, and application data. ADO.NET supports a variety of development needs, including the creation of front-end database clients and middle-tier business objects used by applications, tools, languages, or Internet browsers.

ADO : ActiveX Data Objects **and ADO.Net are two different ways to access database in Microsoft.**

ADO	ADO.Net
ADO is base on **COM** : **Component Object Modelling** based.	ADO.Net is based on **CLR** : **Common Language Runtime** based.
ADO stores data in **binary** format.	ADO.Net stores data in **XML** format i.e. parsing of data.
ADO can't be integrated with XML because ADO have limited access of XML.	ADO.Net can be integrated with XML as having robust support of XML.
In ADO, data is provided by **RecordSet**.	In ADO.Net data is provided by **DataSet** or **DataAdapter**.
ADO is **connection** oriented means it requires continuous active connection.	ADO.Net is **disconnected**, does not need continuous connection.
ADO gives rows as single table view, it scans sequentially the rows using **MoveNext** method.	ADO.Net gives rows as collections so you can access any record and also can go through a table via loop.
In ADO, You can create only **Client** side cursor.	In ADO.Net, You can create both **Client & Server** side cursor.
Using a single connection instance, ADO can not handle multiple transactions.	Using a single connection instance, ADO.Net can handle multiple transactions.

Working with Connection, Command:

Working with Connection:

Process of creating connection:

For SQL Server

Note: Sql Server must be installed

```
String conn_str;
SqlConnection connection;
conn_str = "Data Source=DESKTOP-EG4ORHN\SQLEXPRESS; Initial Catalog=billing;
            User ID=sa;Password=24518300";
connection = New SqlConnection(conn_str);
```

(Here, DESKTOP-EG4ORHN\SQLEXPRESS refers to a data source and billing refers to database name)

Working with Command:

```
SqlConnection connection;
 SqlCommand command;
 String conn_str="Data Source=Raazu\\SQLEXPRESS; Initial
            Catalog=billing; User ID=sa;Password=24518300";
 connection = new SqlConnection(conn_str);
 connection.Open();


String sql = "insert into tblCustomer(name,address) values(“Raaju”,
                    “Birtamode”)";
      command = new SqlCommand(sql, connection);
      command.ExecuteNonQuery();
      connection.Close();
```

For MS-Access

Note: Access Database Engine must be installed

```
OleDbConnection conn;
OleDbCommand command;
  string constr = "Provider=Microsoft.ACE.OLEDB.12.0;Data
        Source=C:\\Users\\Raazu\\Documents\\Visual Studio
        2012\\Projects\\DatabaseTest\\testdb.accdb";
  conn = new OleDbConnection(constr);
  conn.Open();

  string sql = "INSERT INTO tblStudent(name,address)
                    VALUES('Ram','Btm')";
  command = new OleDbCommand(sql,conn);
  command.ExecuteNonQuery();
  Console.WriteLine("Data Inserted Successfully !");
```

DataReader, DataAdaper, Dataset and Datatable :

DataReader is used to read the data from database and it is a read and forward only connection oriented architecture during fetch the data from database. DataReader will fetch the data very fast when compared with dataset. Generally we will use ExecuteReader object to bind data to dataReader.

//Example

```
SqlDataReader sdr = cmd.ExecuteReader();
```

DataReader

- Holds the connection open until you are finished (don't forget to close it!).
- Can typically only be iterated over once
- Is not as useful for updating back to the database

DataSet is a disconnected orient architecture that means there is no need of active connections during work with datasets and it is a collection of DataTables and relations between tables. It is used to hold multiple tables with data. You can select data form tables, create views based on table and ask child rows over relations. Also DataSet provides you with rich features like saving data as XML and loading XML data.

//Example

```
DataSet ds = new DataSet();
  da.Fill(ds);
```

DataAdapter will acts as a Bridge between DataSet and database. This dataadapter object is used to read the data from database and bind that data to dataset. Dataadapter is a disconnected oriented architecture.

//Example

```
SqlDataAdapter sda = new SqlDataAdapter(cmd);
  DataSet ds = new DataSet();
  da.Fill(ds);
```

DataAdapter

- Lets you close the connection as soon it's done loading data, and may even close it for you automatically
- All of the results are available in memory
- You can iterate over it as many times as you need, or even look up a specific record by index
- Has some built-in faculties for updating back to the database.

DataTable represents a single table in the database. It has rows and columns. There is no much difference between dataset and datatable, dataset is simply the collection of datatables.

//Example

```
DataTable dt = new DataTable();
   da.Fill(dt);
```

Difference between DataReader and DataAdapter:

1) A DataReader is an object returned from the ExecuteReader method of a DbCommand object. It is a forward-only cursor over the rows in the each result set. Using a DataReader, you can access each column of the result set, read all rows of the set, and advance to the next result set if there are more than one.
A DataAdapter is an object that contains four DbCommand objects: one each for SELECT, INSERT, DELETE and UPDATE commands. It mediates between these commands and a DataSet though the Fill and Update methods.

2) DataReader is a faster way to retrieve the records from the DB. DataReader reads the column. DataReader demands live connection but DataAdapter needs disconnected approach.

3) Data reader is an object through which you can read a sequential stream of data. it's a forward only data wherein you cannot go back to read previous data. data set and data adapter object help us to work in disconnected mode. data set is an in cache memory representation of tables. the data is filled from the data source to the data set thro' the data adapter. once the table in the dataset is modified, the changes are broadcast to the database back throw; the data adapter.

Complete Example – CRUD operation (MS-Access):

```
using System;
using System.Data;
using System.Data.OleDb;
namespace DatabaseTest
{
    class Program
    {
        OleDbConnection conn;
        OleDbCommand command;
        void CreateConnection()
        {
            string constr = "Provider=Microsoft.ACE.OLEDB.12.0;Data
                  Source=C:\\Users\\Raazu\\Documents\\Visual Studio
                  2012\\Projects\\DatabaseTest\\testdb.accdb";
            conn = new OleDbConnection(constr);
            conn.Open();
        }

        void InsertUpdateDelete(string sql)
        {
            command = new OleDbCommand(sql, conn);
            command.ExecuteNonQuery();
            Console.WriteLine("Operation Performed Successfully !");
        }

        void SelectRecords(string sql)
        {
            command = new OleDbCommand(sql, conn);
            OleDbDataAdapter adapter = new OleDbDataAdapter(command);
```

```
        DataTable dt = new DataTable();
        adapter.Fill(dt);
        if (dt.Rows.Count != 0)
        {
            Console.WriteLine("Sid\t Name\t Address");
            for (int i = 0; i < dt.Rows.Count;i++)
            {
                string sid = dt.Rows[i]["sid"].ToString();
                string name = dt.Rows[i]["name"].ToString();
                string address = dt.Rows[i]["address"].ToString();
                Console.WriteLine(sid+"\t"+name+"\t"+address);
            }
        }
    }

    static void Main(string[] args)
    {
        Program obj = new Program();
        try
        {
            obj.CreateConnection();
            x: Console.WriteLine("1.Insert\t 2.Update\t 3.Delete\t
                    4.Select");
            Console.WriteLine("Enter your choice: ");
            int n = Convert.ToInt32(Console.ReadLine());
            string sql="",nm = "", add = "";
            int id=0;
            switch (n)
            {
                case 1:
                    Console.WriteLine("Ënter Name of Student: ");
                    nm = Console.ReadLine();
                    Console.WriteLine("Ënter Address of Student: ");
                    add = Console.ReadLine();
                    sql = "INSERT INTO tblStudent (name,address)
                            VALUES('"+nm+"','"+add+"')";
                    obj.InsertUpdateDelete(sql);
                    break;
                case 2:
                    Console.WriteLine("Enter id to be updated");
                    id = Convert.ToInt32(Console.ReadLine());
                    Console.WriteLine("Ënter Name of Student: ");
                    nm = Console.ReadLine();
                    Console.WriteLine("Ënter Address of Student: ");
                    add = Console.ReadLine();
                    sql = "UPDATE tblStudent SET name='"+nm+"',
                          address='"+add+"' WHERE sid="+id;
                    obj.InsertUpdateDelete(sql);
                    break;
                case 3:
                    Console.WriteLine("Enter id to be deleted");
                    id = Convert.ToInt32(Console.ReadLine());
                    sql = "DELETE FROM tblStudent WHERE sid="+id;
                    obj.InsertUpdateDelete(sql);
```

```
                    break;
                case 4:
                    sql = "SELECT * FROM tblStudent";
                    obj.SelectRecords(sql);
                    break;
                default:
                    Console.WriteLine("Wrong Choice");
                    break;
            }
            goto x;
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex);
            Console.WriteLine("Connection Failed !");
        }


        Console.ReadKey();
    }
}
```

Connect C# to MySQL

- First make sure you have downloaded and installed the **MySQL Connector/NET** from the MySQL official website.
- Add reference **MySql.Data** in your project.

```
using MySql.Data.MySqlClient;
string constr = "SERVER=localhost; DATABASE=dbtest; UID=root;
                    PASSWORD=;";
MySqlConnection conn = new MySqlConnection(constr);
```

Complete Program for CRUD operation

```
using MySql.Data.MySqlClient;
using System;
using System.Data;
namespace DatabaseTest
{
    class Program
    {
        MySqlConnection conn;
        MySqlCommand command;
        void CreateConnection()
        {
            string constr = "SERVER=localhost; DATABASE=dbtest;
                    UID=root; PASSWORD=;";
            conn = new MySqlConnection(constr);
            conn.Open();
        }
```

```
void InsertUpdateDelete(string sql)
  {
      command = new MySqlCommand(sql, conn);
      command.ExecuteNonQuery();
      Console.WriteLine("Operation Performed Successfully !");
  }

  void SelectRecords(string sql)
  {
      command = new MySqlCommand(sql, conn);
      MySqlDataAdapter adapter = new MySqlDataAdapter(command);
      DataTable dt = new DataTable();
      adapter.Fill(dt);
      if (dt.Rows.Count != 0)
      {
          Console.WriteLine("Sid\t Name\t Address");
          for (int i = 0; i < dt.Rows.Count;i++)
          {
              string sid = dt.Rows[i]["sid"].ToString();
              string name = dt.Rows[i]["name"].ToString();
              string address = dt.Rows[i]["address"].ToString();
              Console.WriteLine(sid+"\t"+name+"\t"+address);
          }
      }
  }

  static void Main(string[] args)
  {
      Program obj = new Program();
      try
      {
          obj.CreateConnection();
          x: Console.WriteLine("1.Insert\t 2.Update\t 3.Delete\t
                                    4.Select");
          Console.WriteLine("Enter your choice: ");
          int n = Convert.ToInt32(Console.ReadLine());
          string sql="",nm = "", add = "";
          int id=0;
          switch (n)
          {
              case 1:
                  Console.WriteLine("Ënter Name of Student: ");
                  nm = Console.ReadLine();
                  Console.WriteLine("Ënter Address of Student: ");
                  add = Console.ReadLine();
                  sql = "INSERT INTO tblStudent (name,address)
                              VALUES('"+nm+"','"+add+"')";
                  obj.InsertUpdateDelete(sql);
                  break;

              case 2:
                  Console.WriteLine("Enter id to be updated");
                  id = Convert.ToInt32(Console.ReadLine());
                  Console.WriteLine("Ënter Name of Student: ");
```

```
                        nm = Console.ReadLine();
                        Console.WriteLine("Ënter Address of Student: ");
                        add = Console.ReadLine();
                        sql = "UPDATE tblStudent SET name='"+nm+"',
                                    address='"+add+"' WHERE sid="+id;
                        obj.InsertUpdateDelete(sql);
                        break;

                    case 3:
                        Console.WriteLine("Enter id to be deleted");
                        id = Convert.ToInt32(Console.ReadLine());
                        sql = "DELETE FROM tblStudent WHERE sid="+id;
                        obj.InsertUpdateDelete(sql);
                        break;

                    case 4:
                        sql = "SELECT * FROM tblStudent";
                        obj.SelectRecords(sql);
                        break;


                    default:
                        Console.WriteLine("Wrong Choice");
                        break;
                }
                goto x;
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
                Console.WriteLine("Connection Failed !");
            }


            Console.ReadKey();
        }
    }
}
```

Complete CRUD operation for SQL Server

```
using System;
using System.Data;
using System.Data.SqlClient;
namespace DatabaseTest
{
    class Program
    {
        SqlConnection conn;
        SqlCommand command;
        void CreateConnection()
        {
            string constr="Data Source=Raazu\\SQLEXPRESS; Initial
                  Catalog=dbtest; User ID=sa;Password=24518300";
```

```
        conn = new SqlConnection(constr);
        conn.Open();
    }

    void InsertUpdateDelete(string sql)
    {
        command = new SqlCommand(sql, conn);
        command.ExecuteNonQuery();
        Console.WriteLine("Operation Performed Successfully !");
    }

    void SelectRecords(string sql)
    {
        command = new SqlCommand(sql, conn);
        SqlDataAdapter adapter = new SqlDataAdapter(command);
        DataTable dt = new DataTable();
        adapter.Fill(dt);
        if (dt.Rows.Count != 0)
        {
            Console.WriteLine("Sid\t Name\t Address");
            for (int i = 0; i < dt.Rows.Count;i++)
            {
                string sid = dt.Rows[i]["sid"].ToString();
                string name = dt.Rows[i]["name"].ToString();
                string address = dt.Rows[i]["address"].ToString();
                Console.WriteLine(sid+"\t"+name+"\t"+address);
            }
        }
    }

    static void Main(string[] args)
    {
        Program obj = new Program();
        try
        {
            obj.CreateConnection();
            x: Console.WriteLine("1.Insert\t 2.Update\t 3.Delete\t
                                    4.Select");
            Console.WriteLine("Enter your choice: ");
            int n = Convert.ToInt32(Console.ReadLine());
            string sql="",nm = "", add = "";
            int id=0;
            switch (n)
            {
                case 1:
                    Console.WriteLine("Ënter Name of Student: ");
                    nm = Console.ReadLine();
                    Console.WriteLine("Ënter Address of Student: ");
                    add = Console.ReadLine();
                    sql = "INSERT INTO tblStudent (name,address)
                                VALUES('"+nm+"','"+add+"')";
                    obj.InsertUpdateDelete(sql);
                    break;
```

```
                    case 2:
                        Console.WriteLine("Enter id to be updated");
                        id = Convert.ToInt32(Console.ReadLine());
                        Console.WriteLine("Ënter Name of Student: ");
                        nm = Console.ReadLine();
                        Console.WriteLine("Ënter Address of Student: ");
                        add = Console.ReadLine();
                        sql = "UPDATE tblStudent SET name='"+nm+"',
                                    address='"+add+"' WHERE sid="+id;
                        obj.InsertUpdateDelete(sql);
                        break;

                    case 3:
                        Console.WriteLine("Enter id to be deleted");
                        id = Convert.ToInt32(Console.ReadLine());
                        sql = "DELETE FROM tblStudent WHERE sid="+id;
                        obj.InsertUpdateDelete(sql);
                        break;

                    case 4:
                        sql = "SELECT * FROM tblStudent";
                        obj.SelectRecords(sql);
                        break;


                    default:
                        Console.WriteLine("Wrong Choice");
                        break;
                }
                goto x;
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
                Console.WriteLine("Connection Failed !");
            }


            Console.ReadKey();
        }
    }
}
```

Writing Windows Form Applications

Introduction to Win Forms:

Windows Forms (WinForms) is a graphical (GUI) class library included as a part of Microsoft .NET Framework, providing a platform to write rich client applications for desktop, laptop, and tablet PCs.

A Windows Forms application is an event-driven application supported by Microsoft's .NET Framework. Unlike a batch program, it spends most of its time simply waiting for the user to do something, such as fill in a text box or click a button.

All visual elements in the Windows Forms class library derive from the Control class. This provides a minimal functionality of a user interface element such as location, size, color, font, text, as well as common events like click and drag/drop.

Basic Controls:

The following table lists some of the commonly used controls:

S.N.	Widget & Description
1	**Forms** The container for all the controls that make up the user interface.
2	**TextBox** It represents a Windows text box control.
3	**Label** It represents a standard Windows label.
4	**Button** It represents a Windows button control.
5	**ListBox** It represents a Windows control to display a list of items.
6	**ComboBox** It represents a Windows combo box control.
7	**RadioButton**

	It enables the user to select a single option from a group of choices when paired with other RadioButton controls.
8	**CheckBox** It represents a Windows CheckBox.
9	**PictureBox** It represents a Windows picture box control for displaying an image.
10	**ProgressBar** It represents a Windows progress bar control.
11	**ScrollBar** It Implements the basic functionality of a scroll bar control.
12	**DateTimePicker** It represents a Windows control that allows the user to select a date and a time and to display the date and time with a specified format.
13	**TreeView** It displays a hierarchical collection of labeled items, each represented by a TreeNode.
14	**ListView** It represents a Windows list view control, which displays a collection of items that can be displayed using one of four different views.

For more information, refer to the practical exercises.

Web Applications using ASP.NET

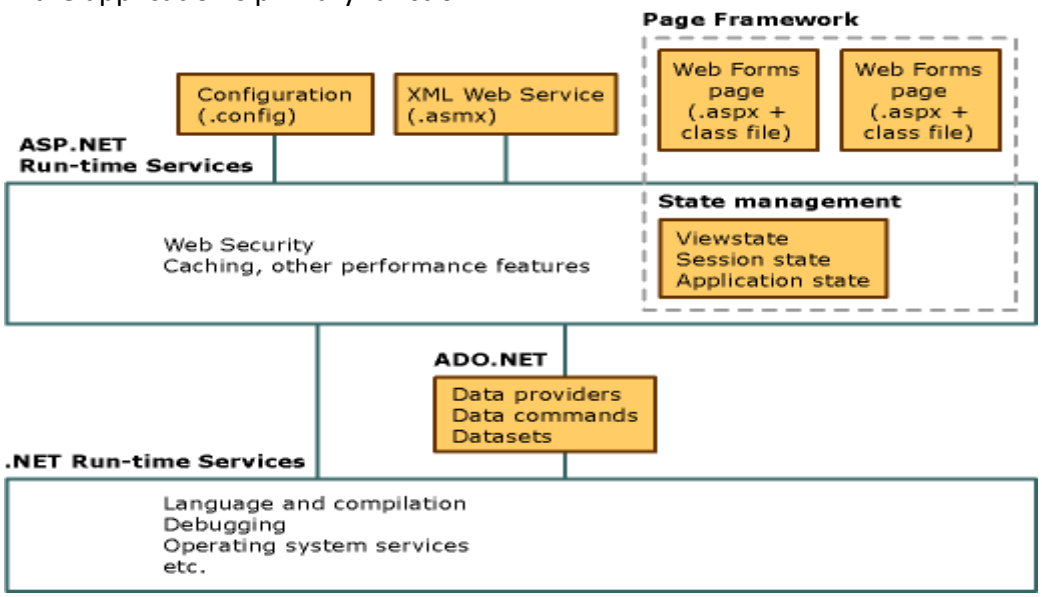
A Visual Studio Web application is built around ASP.NET. ASP.NET is a platform — including design-time objects and controls and a run-time execution context — for developing and running applications on a Web server.

ASP.NET Web applications run on a Web server configured with Microsoft Internet Information Services (IIS). However, you do not need to work directly with IIS. You can program IIS facilities using ASP.NET classes, and Visual Studio handles file management tasks such as creating IIS applications when needed and providing ways for you to deploy your Web applications to IIS.

Elements of ASP.NET Web Applications

Creating ASP.NET Web applications involves working with many of the same elements you use in any desktop or client-server application. These include:

- **Project management features** When creating an ASP.NET Web application, you need to keep track of the files you need, which ones need to be compiled, and which need to be deployed.
- **User interface** Your application typically presents information to users; in an ASP.NET Web application, the user interface is presented in Web Forms pages, which send output to a browser. Optionally, you can create output tailored for mobile devices or other Web appliances.
- **Components** Many applications include reusable elements containing code to perform specific tasks. In Web applications, you can create these components as XML Web services, which makes them callable across the Web from a Web application, another XML Web service, or a Windows Form, for example.
- **Data** Most applications require some form of data access. In ASP.NET Web applications, you can use ADO.NET, the data services that are part of the .NET Framework.
- **Security, performance, and other infrastructure features** As in any application, you must implement security to prevent unauthorized use, test and debug the application, tune its performance, and perform other tasks not directly related to the application's primary function.



Different Types of form controls in ASP.NET

Button Controls

ASP.NET provides three types of button control:

- **Button** : It displays text within a rectangular area.
- **Link Button** : It displays text that looks like a hyperlink.
- **Image Button** : It displays an image.

```
<asp:Button ID="Button1" runat="server" onclick="Button1_Click"
Text="Click" / >
```

Text Boxes and Labels

Text box controls are typically used to accept input from the user. A text box control can accept one or more lines of text depending upon the settings of the TextMode attribute.

Label controls provide an easy way to display text which can be changed from one execution of a page to the next. If you want to display text that does not change, you use the literal text.

```
<asp:TextBox ID="txtstate" runat="server" ></asp:TextBox>
```

Check Boxes and Radio Buttons

A check box displays a single option that the user can either check or uncheck and radio buttons present a group of options from which the user can select just one option.

To create a group of radio buttons, you specify the same name for the **GroupName** attribute of each radio button in the group. If more than one group is required in a single form, then specify a different group name for each group.

If you want check box or radio button to be selected when the form is initially displayed, set its Checked attribute to true. If the Checked attribute is set to true for multiple radio buttons in a group, then only the last one is considered as true.

```
<asp:CheckBox ID= "chkoption" runat= "Server">
</asp:CheckBox>
```

```
<asp:RadioButton ID= "rdboption" runat= "Server">
</asp: RadioButton>
```

List box Control

These control let a user choose from one or more items from the list. List boxes and drop-down lists contain one or more list items. These lists can be loaded either by code or by the **ListItemCollection** editor.

```
<asp:ListBox ID="ListBox1" runat="server">
</asp:ListBox>
```

HyperLink Control

The HyperLink control is like the HTML <a> element.

```
<asp:HyperLink ID="HyperLink1" runat="server">
   HyperLink
</asp:HyperLink>
```

Image Control

The image control is used for displaying images on the web page, or some alternative text, if the image is not available.

```
<asp:Image ID="Image1" ImageUrl="url" runat="server">
```

Drop down List Control

```
<asp:DropDownList ID="DropDownList1" runat="server"
</asp:DropDownList>
```

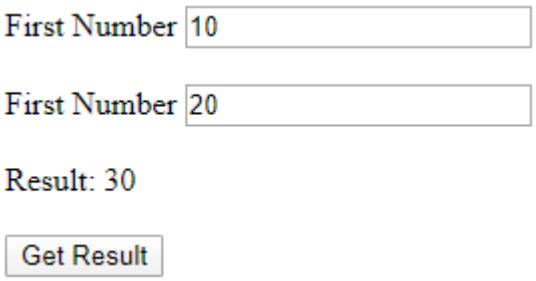
Launch another form on button click

```
protected void btnSelect_Click(object sender, EventArgs e)
        {
            Response.Redirect("AnotherForm.aspx");
        }
```

Example 1 – Creating a basic form in ASP.Net


```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="MyForm.aspx.cs" Inherits="WebApplication1.MyForm" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>This is my first application</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:Label ID="lblName" runat="server" Text="Name:"></asp:Label>
        <asp:TextBox ID="txtName" runat="server"></asp:TextBox><br/>
        <asp:Label ID="lblAddress" runat="server" Text="Address:">
                  </asp:Label>
        <asp:TextBox ID="txtAddress" runat="server"></asp:TextBox> <br/>
        <asp:Button ID="btnSubmit" runat="server" Text="Submit" />
    </form>
</body>
</html>
```

Example – 2 (Handling Events)



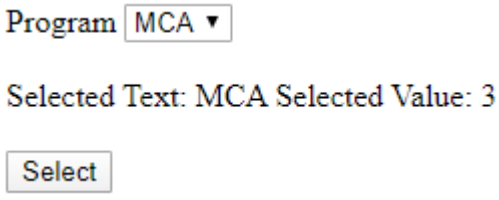
EventHandling.aspx

```
<form id="form1" runat="server">
    <div>
        <asp:Label ID="Label1" runat="server" Text="First Number">
            </asp:Label>
        <asp:TextBox ID="txtFirst" runat="server"></asp:TextBox>
                        <br/><br/>
        <asp:Label ID="Label2" runat="server" Text="First Number">
                    </asp:Label>
        <asp:TextBox ID="txtSecond" runat="server"></asp:TextBox>
                    <br/><br/>
        <asp:Label ID="lblResult" runat="server" Text="Result:">
                    </asp:Label> <br/><br/>
        <asp:Button ID="btnSubmit" runat="server" Text="Get Result"
                    onClick="btnSubmit_Click"/>
    </div>
    </form>
```

EventHandling.cs

```
protected void btnSubmit_Click(object sender, EventArgs e)
        {
            int first = Convert.ToInt32(txtFirst.Text);
            int second = Convert.ToInt32(txtSecond.Text);
            int res = first + second;
            lblResult.Text = "Result: " + res;
        }
```

Example 3 – Using Drop down list



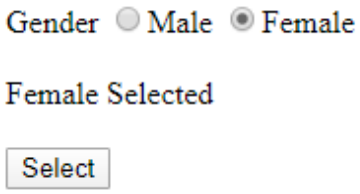
Dropdown.aspx

```
<form id="form1" runat="server">
    <div>
        <asp:Label ID="Label1" runat="server" Text="Program">
            </asp:Label>
        <asp:DropDownList ID="dropProgram" runat="server">
                  </asp:DropDownList>
        <br/><br/>
        <asp:Label ID="lblSelected" runat="server" Text="Selected:">
            </asp:Label>
        <br/><br/>
        <asp:Button ID="btnSelect" runat="server" Text="Select"
            OnClick="btnSelect_Click" />
    </div>
    </form>
```

Dropdown.cs

```
private void LoadData()
        {
            List<ListItem> mylist=new List<ListItem>();
            mylist.Add(new ListItem("BCA","1"));
            mylist.Add(new ListItem("BBA","2"));
            mylist.Add(new ListItem("MCA","3"));
            mylist.Add(new ListItem("MBA","4"));
            dropProgram.Items.AddRange(mylist.ToArray());
        }

protected void btnSelect_Click(object sender, EventArgs e)
        {
            string text = dropProgram.SelectedItem.ToString();
            string value = dropProgram.SelectedValue;
            lblSelected.Text = "Selected Text: " + text + " Selected
                  Value: " + value;
        }
```

Example – 4 Using Radio button



example.aspx

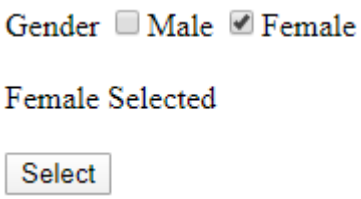
```
<form id="form1" runat="server">
    <div>
        <asp:Label ID="Label1" runat="server" Text="Gender"></asp:Label>
        <asp:RadioButton ID="radioMale" Text="Male" GroupName="gender"
                  runat="server" />
        <asp:RadioButton ID="radioFemale" Text="Female"
                  GroupName="gender" runat="server" />
        <br/><br/>
        <asp:Label ID="lblSelected" runat="server" Text="Selected
```

```
            Radio:"> </asp:Label>
        <br/><br/>
        <asp:Button ID="btnSelect" runat="server" Text="Select"
                OnClick="btnSelect_Click" />
    </div>
    </form>
```

example.cs

```
protected void btnSelect_Click(object sender, EventArgs e)
        {
            if (radioMale.Checked)
                lblSelected.Text = "Male Selected";
            else
                lblSelected.Text = "Female Selected";
        }
```

Example – 5 Using Check box



example.aspx

```
<form id="form1" runat="server">
    <div>
        <asp:Label ID="Label1" runat="server" Text="Gender"></asp:Label>
        <asp:CheckBox ID="chkMale" Text="Male" GroupName="gender"
                runat="server" />
        <asp:CheckBox ID="chkFemale" Text="Female"
                GroupName="gender" runat="server" />
        <br/><br/>
        <asp:Label ID="lblSelected" runat="server" Text="Selected
            Radio:"> </asp:Label>
        <br/><br/>
        <asp:Button ID="btnSelect" runat="server" Text="Select"
                OnClick="btnSelect_Click" />
    </div>
    </form>
```

example.cs

```
protected void btnSelect_Click(object sender, EventArgs e)
        {
            if (chkMale.Checked)
                lblSelected.Text = "Male Selected";
            else
                lblSelected.Text = "Female Selected";
        }
```

Validation Controls in ASP.NET

An important aspect of creating ASP.NET Web pages for user input is to be able to check that the information users enter is valid. ASP.NET provides a set of validation controls that provide an easy-to-use but powerful way to check for errors and, if necessary, display messages to the user.

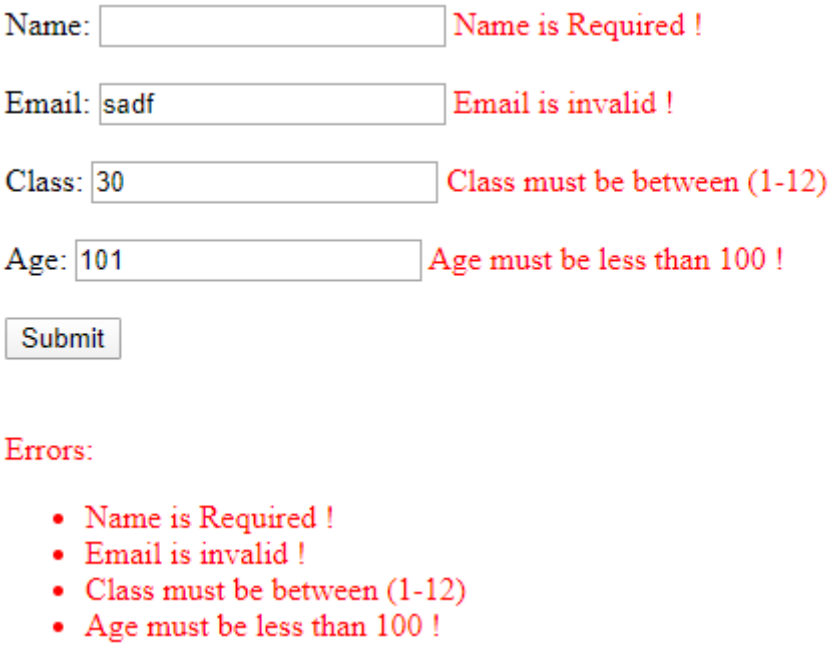
There are six types of validation controls in ASP.NET

- RequiredFieldValidation Control
- CompareValidator Control
- RangeValidator Control
- RegularExpressionValidator Control
- CustomValidator Control
- ValidationSummary

The below table describes the controls and their work:

Validation Control	Description
RequiredFieldValidation	Makes an input control a required field
CompareValidator	Compares the value of one input control to the value of another input control or to a fixed value
RangeValidator	Checks that the user enters a value that falls between two values
RegularExpressionValidator	Ensures that the value of an input control matches a specified pattern
CustomValidator	Allows you to write a method to handle the validation of the value entered
ValidationSummary	Displays a report of all validation errors occurred in a Web page

Example of Validation Controls



```
<form id="form1" runat="server">
    <div>
        <asp:Label ID="Label1" runat="server" Text="Name:"></asp:Label>
        <asp:TextBox ID="txtName" runat="server"></asp:TextBox>
        <asp:RequiredFieldValidator ID="validator1" runat="server"
            ForeColor="Red" ErrorMessage="Name is Required !"
            ControlToValidate="txtName"> </asp:RequiredFieldValidator>
        <br/><br/>

        <asp:Label ID="Label2" runat="server" Text="Email:"></asp:Label>
        <asp:TextBox ID="txtEmail" runat="server"></asp:TextBox>
        <asp:RegularExpressionValidator ID="validator2" runat="server"
            ForeColor="Red" ControlToValidate="txtEmail"
                  ErrorMessage="Email is invalid !"
         ValidationExpression="\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-
                                    .]\w+)*">
            </asp:RegularExpressionValidator>
        <br/><br/>

        <asp:Label ID="Label3" runat="server" Text="Class:"></asp:Label>
        <asp:TextBox ID="txtClass" runat="server"></asp:TextBox>
        <asp:RangeValidator ID="validator3"
               runat="server" ControlToValidate="txtClass"
                        ForeColor="Red"
               ErrorMessage="Class must be between (1-12)"
                             MaximumValue="12"
               MinimumValue="1" Type="Integer">
            </asp:RangeValidator>
         <br/><br/>

        <asp:Label ID="Label4" runat="server" Text="Age:"></asp:Label>
        <asp:TextBox ID="txtAge" runat="server"></asp:TextBox>
        <asp:CompareValidator ID="validator4" runat="server"
                  ValueToCompare="100"   ControlToValidate="txtAge"
                  ErrorMessage="Age must be less than 100 !"
                ForeColor="Red" Operator="LessThan" Type="Integer">
        </asp:CompareValidator>
        <br/><br/>

        <asp:Button ID="btnSubmit" runat="server" Text="Submit" />
        </div>
        <br/><br/>

      <asp:ValidationSummary ID="validator5" runat="server"
            ForeColor="Red" DisplayMode ="BulletList"
            ShowSummary ="true" HeaderText="Errors:" />
    </form>
```